UNIVERSITY OF OSLO
Department of Informatics

Tool-based Method for Testing Policy Adherence

Master thesis

Monika Sperstad Køller

1st May 2012

Acknowledgment

I would like to show my gratitude to my supervisor, **Ketil Stølen**, whose guidance and support helped me through the work of this project. I would also like to show my gratitude to my co-supervisors **Mass Soldal Lund**, **Bjørnar Solhaug** and **Fredrik Seehusen**, who helped me understand the subjects of this project.

My fellow students at the ninth floor, who held out with my chatter and high thinking of the problems, also need to get a thank you. You all kept me motivated, and helped my as far as you could. I also would like to thank my family, boyfriend and friends, who had to listen to my problems when I got stuck, and for being there for me and keeping me motivated.

Abstract

The purpose of this project has been to look at different approaches to test if a software program works as wanted or as expected. Testing is a broad term, and is used in different ways and for different purposes. This project has a look at how to test if a software program follows rules specifying requirements in system behavior. The focus has been to test in a manner that was accurate and efficient.

This project is based on work that has been done on policy-based management and on testing of sequence diagrams. Policy-based management has gotten increased attention the last decade, and this project used policies to define the rules of the software program. Sequence diagrams are diagrams that are used to model interactions, and are in this manner suitable for defining or specifying both policies and software programs. This project addresses the issue of generating test diagrams based on sequence diagrams, and then conduct testing with them.

Based on an evaluation of a tool, which address some of the questions in this project, we developed a method for testing a software specification against a policy specification. The method does the testing by generating test diagrams, which are later applied to the software specification by running tests. We also developed a tool to support the method.

From the evaluation in this project the artifacts we developed are solving some of the issues discussed in this project, but there is also room for improvements. Testing is a time-consuming task, and there is a need to work further with our artifacts in order for them to be employed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When we want to make a product, like a cup, a jacket or a software program, we want to make sure that the product works as expected. We want a cup to be able to contain liquid, and that a person can drink the liquid from it. To find out whether the cup fulfills such requirements, we can test it. One test can be to put liquid into the cup, and see if the liquid stays there. Another test can be to see if a person manages to drink the liquid from the cup. If the cup passes both tests, we have substantiated that the product fulfills the requirements and works as expected. Testing a cup is relatively easy. A cup has been used for a long time. We therefore know a lot of what design it can have, and the requirements we have to it. It is the same for a jacket.

When it comes to software programs, and software testing, the history is shorter [34]. Different software programs are often very different from each other, which mean that requirements and tests vary between different software programs. It might be that we can not make all the tests that are needed in order to test all the requirements of a software program. It might also be that we can not come up with all the requirements that we would want to test [29].

When we want to create a software program, the first thing we do is to define its requirements to describe what it should do and what it should not do. This is often done based on input from all the stakeholders to the software program, in order to capture requirements with respect to all the different roles that will interact, or get affected by the software program being made.

The software program is then developed, and during the development process we may test it to see whether it fulfills the requirements. This is obviously a compressed description of the development process, but still captures essential elements of the iterative development process of requirements capturing, development and testing. These elements are moreover

main topics of this project.

The more complex the software program is the more can go wrong during the development, and the more we have to test. It is relatively easy to test if a software program does what it should if the data sent to it are *correct* and the tester interacts with the software program in the *correct* way. Correct here meaning the expected data, and the expected way of interaction.

On the other hand it is difficult to test if a software program does exactly what it should if unexpected things happen. We might never be sure we have tested all possible data send to it, or tested all the possible scenarios that can happen while interacting with the software program. It might also be too time-consuming to test all the possible scenarios even if we knew them [16]. To be sure that a software program is totally correct, we have to test all possible scenarios.

There has been done a lot of work on testing, for example on how it can be done as easy and accurately as possible, and within a short time-interval. In this project we look at, and use, some of the techniques that have been defined.

The focus on security in systems, like software programs, has become more important now as we use technology more often. Today we "offer, consume and depend on electronic services provided over computerized networks such as the Internet" [45]. We put a lot more information into systems and we trust them. When we depend on something as the Internet, and put a lot of data there, we have to make sure that it is secured. We have to make sure that unauthorized people do not get access to the data, and that authorized people do.

The computerized systems of today may considerably impact the lives of humans, companies, and the environment [16]. Some examples are systems in hospitals and banks. If a system in a hospital does not do as expected, the consequences can be big for the human who is interacting with it, or are depending on it. It is also the aspect of the system giving unauthorized people access to protected information, and not giving access to authorized people. This can be important especially for systems that communicate with databases keeping information for governments, the military, banks, etc.

We have different examples of disasters that have happened, which could have been avoided if the software program had been through more tests, such as the *20 Famous Software Disasters* [33], and five disasters in the book [16] page 9. One example is described in both, and tells about *Ariane 5's* first test flight. It had an error in its software and was destructed less then a minute after departure. As it is stated in the book "[i]f every possible test had been run, the problem would have been detected. However, if every test had been run, the testing would still be running now" [16]. This underlines the problem that testing is time consuming, but nevertheless crucial in developing. The book also stated that the

right sort of tests where not conducted, and then we come back to the problem of finding good requirements.

The perfect situation to test a system would be that we had a perfect tool, and all the requirements needed for testing the system. The perfect tool would be a tool that took the system specification, and the requirements, defined as rules, as input, and in a short time-interval tested whether the system followed all those rules, or not. It would answer yes or no, and the answer would tell the tester if the system worked as expected or not respectively. It would also give a short explanation for the answer, so the tester would know where the answer came from. The explanation would tell which part of the system that followed the rules, or not followed the rules. The perfect tool will not be built here, and we do not have intentions in trying to build it.

Testing a system can be done in many ways. In this project we do it by using policies. A policy specifies a set of rules, and we want to test if a system adheres to those rules. Policy-based management have gotten increased attention the last decade [39] [41], and has "emerged as an adaptive and flexible approach to administer and control distributed systems with respect to issues as security, access control, services, networks and trust" [45]. "An important motivation for the use of policies for system management is that they allow systems to be dynamically changed in order to meet new or different requirements, without stopping the system and without changing the underlying implementation" [45]. So this makes it easier to define new rules, change the existing ones, and remove existing ones.

1.1 Contributions of this Project

The result from this project contributes to the problem of testing systems with respect to the system requirements. In order to capture and represent system requirements we have based the project on existing work on policy-based management, by specifying requirements as policy rules constraining system behavior. The system under test, on the other hand, has in the project been represented as a system specification describing the system behavior. Hence, the overall problem addressed in this project is how to test policy adherence by testing the system specification together with the policy specification.

The contributions of this project are two main artifacts, namely a method for testing policy adherence and a tool to support the method. The method utilize existing techniques for testing software programs and provides a systematic, stepwise approach to support system developers, testers and policy developers in developing software programs that fulfill the identified system requirements. The tool is developed to support the method by automating several of the tasks the method consists of.

1.2 Overview of Chapters

In Chapter 2 we give some background to the work of this project. We discuss policies, and how policies may be defined and specified. We also introduce the notion of *adherence*, capturing what it means that a system satisfies or fulfills a set of policy rules. We will moreover look at testing, and see how we can conduct policy adherence testing.

In Chapter 3 we describe in more details the problems addressed by this project, and give the reason for our choice of research method. We also describe how the work on this project was planned, and give a description of the steps we conducted to implement the plan.

Chapter 4 gives the characterizing of the needs of this project, in particular by identifying the stakeholders of the artifacts developed in this project, and the requirements to the artifacts given their stakeholders.

In Chapter 5 we give an overview of the state of the art. We look at modeling in general and different modeling languages that can be used for our purpose. We also have a look at model based testing. The last section in this chapter will have a look at existing tools that support defining sequence diagrams. We also present a tool we can use in order to find requirements to our artifacts.

In Chapter 6 we present two different approaches to conduct policy adherence testing, and evaluate each of them. We also evaluate the Escalator, and see how it can support the two approaches.

Chapter 7 presents the developed artifacts, and Chapter 8 contains the evaluation of our artifacts. The evaluation is conducted with respect to the requirements we have defined in Chapter 4.

In the last chapter, Chapter 9, we try to wrap up the work, and discuss this project shortly. Since this project had a short duration, we where not able to make the artifacts so that they included all functionality wanted. Therefore, in this chapter, we also go through what needs to be improved and further developed in future work.

Chapter 2

Background

In this chapter we give some background to the work of this project. Section 2.1 is about policies, what they are and how to specify them. In Section 2.2 we describe the notion of *adherence*. Section 2.3 is about testing. We look at how to conduct policy adherence testing with the use of sequence diagrams, but before that give a short explanation of what a sequence diagram is.

2.1 Policies

As stated in the introduction we can look at a policy as a set of rules. Each rule describes a scenario that must, must not, or should be allowed to happen while a system, such as a software program, is running. What the policy describes is requirements to system behavior during its execution.

There are many different ways of specifying a policy. One way to specify a policy is to "explicitly define *requirements* on the system execution" [19]. This approach says that the system has to run in a way that satisfies the requirements. Another way is an approach that "specifies exclusive *rights* to execute given actions under specific conditions" [19]. This approach says that if the system runs this way it can do this, and if it runs that way it can do that. There are also combinations of these two approaches, as well as others.

This project builds on an existing approach to policy-based management using sequence diagrams [45]. We attempt to further develop this work, unless we see things that we have to change to achieve the objectives of this project.

We specify a policy as a UML sequence diagram [4] [55], which contain both a trigger and a body. The trigger specifies the condition under which the policy rule applies, and the body specifies the behavior that must, must not, or should be allowed if the trigger is true. The

specification of a policy rule by a trigger and body is supported by the policy specification language presented in [45], and the language is referred to as Deontic STAIRS. A distinction is moreover made between event triggers and state triggers. "An event triggered policy applies by the occurrence of a given event, whereas a state triggered policy applies in a given set of states" [45], and there are policy triggers that are a combination of these. The trigger must specify which actor the rule should apply to. This actor is referred to as the addressee of the policy [45]. Figure 2.1, adopted from [45], shows how we structure a policy.

A rationale for using sequence diagrams for policy specification is that "since policies express constraints on behavior, sequence diagrams are a suitable candidate for policy specification" [45]. Sequence diagrams also have some benefits regarding communication with stakeholders of different backgrounds do to their "intuitive and easy to understand representations of interactions" [45].

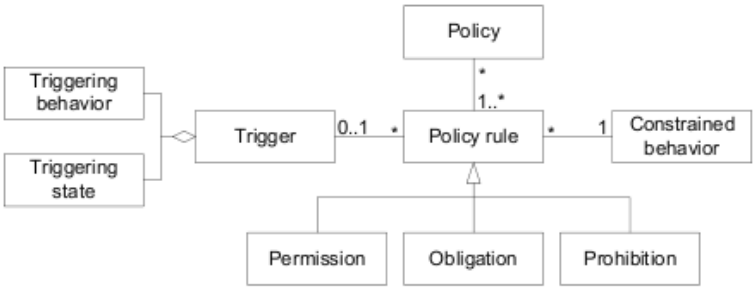


Figure 2.1: The structure of policies [45] p.155

There are three different policy rules supported by Deontic STAIRS, namely obligation, permission and prohibition. The first one, obligation, states that if the "policy trigger executes or applies, the addressee is required to conduct the behavior" [45]. The second one, permission, states that if the "policy trigger executes or applies, the addressee is allowed to conduct the behavior" [45]. The third one, prohibition, states that if the "policy trigger executes or applies, the addressee is forbidden from conducting the behavior" [45].

With the use of these policies we can test if a system does what it should in a certain situation (obligation), that something never will happen in another situation (prohibition), and that there is an alternative that something can happen in another situation (permission). For example, we can test if authorized people will get access if they are logged in with the right information, and that unauthorized do not get access since they are not logged in with the right information. This is done by testing a system specification together with a policy specification.

2.2 Adherence

"Adherence of a system to a policy specification means that the system (implementation) satisfies the policy." [45]. Adherence is the relation between a system and its policies.

For a policy to be enforceable it cannot be inconsistent or conflicting. This means that the system can not adhere to two policies saying the opposite of each other. For instance there cannot be two policies applying to the system that have the same trigger and body, where one is an obligation and the other one a prohibition. That will make the policies conflicting with each other.

"Adherence to a policy means to satisfy or fulfill the policy" [45]. In order to have adherence between the system and its policies the system has to behave like specified in the policies. If a policy-trigger is triggered, then the system, depending on the kind of rule, must, must not or should be allowed to have the traces of the policy-body. If not there is conflict between the system and its policy.

So adherence is a term capturing what it means that a system specification satisfies a policy specification. This is our focus in this project. We want to develop a method for testing policy adherence.

2.3 Testing

The perfect tool introduced in Chapter 1 would have the functionality to test if a system adheres to some existing policies. In this project we have both the system and the policies defined as sequence diagrams. In this section we first discuss testing in general, and thereafter describe what a sequence diagram is, and how we can test policy adherence with the specifications defined as sequence diagrams.

2.3.1 Testing in General

Testing a system provides objective information about how the system is actually behaving [53]. Testing helps determine not only whether a system behaves as it should, but also how well it behaves [29]. Different tests will give different answers, and it is important to know what to test. One kind of test is to find out if the system does what is expected. But we can also test how much time the system uses to perform a task, or how user-friendly it is. All of these different kinds of tests have to be conducted in different ways.

We can split the testing of a system into several categories, and below we have listed four of them. These four we have from [30].

- Unit Testing

Test the basic units of software.

- Integration Testing

  Test the communication between two or more units.

- System Testing

  Test the system as a whole, where the functionality is in focus.

- Acceptance Testing

  Is being conducted after the product is delivered, and here the purpose it to "give confidence that the system is working" [30].

These four different kinds of tests usually are conducted at different stages in a development process. Most often in the order they are stated here.

There are also two basic approaches when testing a system, namely black-box testing and white-box testing [3]. Black-box testing is focusing on what the system under test gives as output, when given input. It does not test any *internal* function, but the result the system gives. On the other hand we have white-box testing that focus on testing the *inside* of the system under test. These two approaches are broader explained in [3].

In this project we aim at providing a method for finding out if a system adheres to some policies, in other words if the system runs in a way that is not conflicting with any policies. There are different approaches we can use when testing this. Since we made a choice to represent the system specification and the policy specification as sequence diagrams it is a natural choice to conduct the testing with the use of these sequence diagrams. In the next sub-section we have a look at how to conduct testing with sequence diagrams.

2.3.2 Policy Adherence Testing

Sequence diagrams are a type of UML interaction diagrams [4]. Interaction diagrams are diagrams that show an interaction between system objects, with messages to and from those objects [4].

"A sequence diagram is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordering in increasing time, along the Y axis" [4]. So the main things sequence diagrams contain are objects, and messages send to and from those objects, ordered in time.

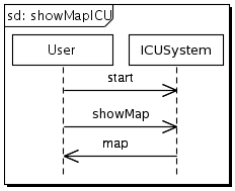


Figure 2.2: A sequence diagram

We have illustrated a sequence diagram in Figure 2.2. The sequence diagram contain two objects; User and ICUSystem, and three messages; start, showMap, and map. The first two messages are sent from User to ICUSystem, and the last one from ICUSystem to User.

Each of the messages has two events, the first event is sending the message, and the second is receiving the message. A trace of a sequence diagram represents a system run, and is a sequence of events in the order they happen in the system run [17]. The system represented in the sequence diagram in Figure 2.2 contains two traces: <!start, ?start, !showMap, ?showMap, !map, ?map>, and <!start, !showMap, ?start, ?showMap, !map, ?map>. Where the symbol ! represents sending a message and the symbol ? represents receiving a message.

In order to test the behavior in a sequence diagram, we need to analyze the messages sent and received. We need to know who sends and who receives what messages, and in what order. We can also use a approach that are more similar to black-box testing, where we give some input, and see if we get the desired output.

When testing policy adherence between two sequence diagrams, one system diagram and one policy diagram, we need to test if one trace from the trigger of the policy is present in a trace from the system. If it is we also need to test if a trace from the body of the policy is present in the same trace from the system. From this we see that in order to test policy adherence, with the use of sequence diagrams, we need to do testing with the events that the sequence diagrams contain.

Testing policy adherence with respect to permission rules is outside the scope of this project, we will only address the rules of obligation and prohibition. There are two reasons for that. First, testing policy adherence with respect to permission rules is more complicated than the other rules, and we decided to first deal with the most basic problems of policy adherence testing. The reason for the higher complexity is that while adherence to obligation rules and prohibition rules can be determined by examining traces one at a time, adherence to permission rules can be determined only by examining sets of traces. With permission rules we have to examining sets of traces since the rule require an alternative to be offered

as potential behavior.

Second, in order to capture requirements to potential behavior as specified by permission rules there is a need for expressiveness that is not supported by the UML standard [55] [9]. The sequence diagrams used in this project are defined using UML standard only. To get further details on the problem of adherence to permission rules consult [39].

Chapter 3

Research Method

This project aims at developing a tool-based method for testing policy adherence. We want the method to facilitate the testing by generating test diagrams from both a system specification and a policy specification. It should then test those test diagrams together with the system specification, and conclude with a result that tells if the system specification adheres to the policy specification. In addition to telling if the system specification adheres to the policy specification, we want the result to give a reason that justifies the statement.

Because this project is concerned with developing new artifacts, we look to developments models. There are many different development models we can use in this kind of projects. We have illustrated some of these models in Figure 3.1 on page 12 and Figure 3.2 on page 13.

They all differ, but have the same goal; to find a product's requirements, make its design, implement it, and deliver it to the client. The way the different steps are performed, and how many times however, vary.

The Waterfall model [15], illustrated in Figure 3.1 on page 12, is a structural system development model with five steps. We start by finding the requirements, and then go down the steps. When we go down one step we cannot go back to previous steps. When using this model, we have to be sure that we spend enough time in each step. This model is not adapted for the situation where the requirements can change in the process.

This model is not that much used, and there has been made many different versions of this model, to improve it. The other versions allow us to go back to previous steps if there are needs for it.

There is also a model called Methodology for Algorithmic Problem Solving (MAPS) [51], which also is illustrated in Figure 3.1 on page 12. It has similarities to the Waterfall method, but has more steps. This method starts with understanding the problem to be solved, by working with a problem statement. The next is to define input that can be given to the

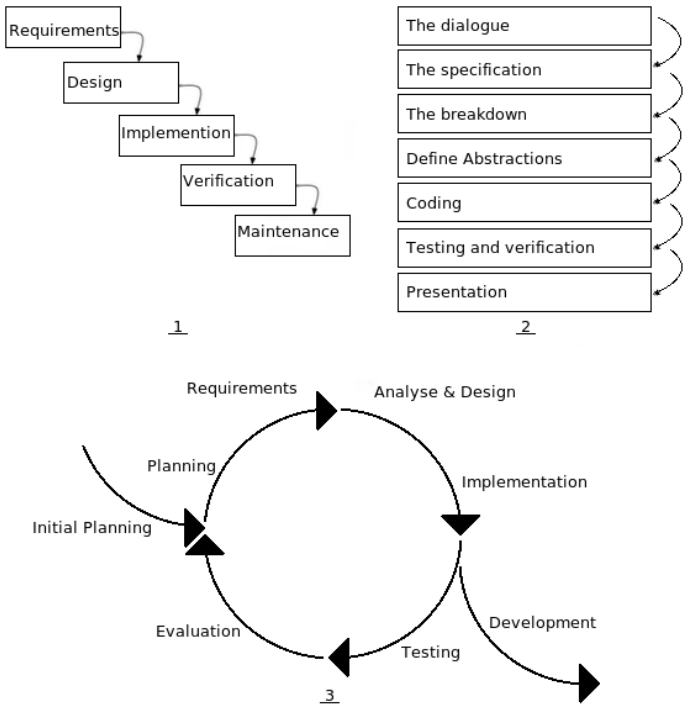


Figure 3.1: 1: Waterfall model 2: MAPS [51] 3: Iterative and Incremental model [2]

product, and state the desired output for each input. Stage three are breaking down the problem in different parts, and planning how to solve each part. Then we try to see if any of the parts have been made before, so that we can reuse previous work. In the fifth step the coding starts and we implement each part. After all parts are implemented, we start testing and verifying the product. We test by giving the product all the different inputs defined in step two, and see if the desired output is produced. The last step is to present the product to the client.

We also have an illustration of the Iterative and Incremental model, illustrated in Figure 3.1. With this model we make the design for the product first, but develop part after part, and evaluate each part when it is made. This can be done as an iterative process, where we work with one part over again if we see problems or possibilities to improve it. The design can also be changed later in the process, also as an iterative process.

The last model we are going to have a look at is the Spiral model, illustrated in Figure 3.2. This model can be said to be a combination of the Waterfall model and the Iterative and Incremental model. With this model we go as a spiral around the product, and for each round add more functionality. Each round starts with a design goal, and ends with the user

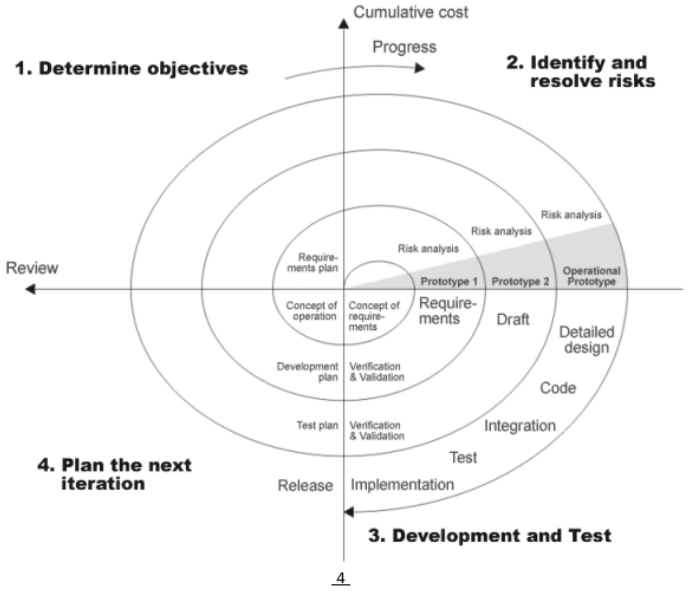


Figure 3.2: 4: Spiral model [54]

evaluating the product thus far.

In our project we have a mixture of the MAPS, and the Iterative and Incremental model. We also change some of the steps, and remove some. The steps we have in our project are described in Table 3.1 on page 14.

3.1 Characterize the Needs

After defining the project, we need to define the stakeholders, and find the requirements. Our tool will not be a complete tool, but the first version of a future complete tool. When we find our stakeholders we will have that in mind, but also think that we need to develop a tool that works as possible future stakeholders would want it to. The importance of finding the stakeholders and have a communication with them are many. One of the main benefits is that "[y]ou can use the opinions of the most powerful stakeholders to shape your projects at an early stage. Not only does this make it more likely that they will support you, their input can also improve the quality of your project" [49].

When we have defined our stakeholders, we can begin to find requirements to the artifacts. We make requirements to the method, and the tool. We also make requirements for the different roles the tool will contain in order to have more detailed requirements.

Step	Description
Characterize the needs	The first phase is to write a problem analysis that will contain an introduction to the project, the background we need, and the state of the art. These subjects are found respectively in Chapter 1, 2, and 5. We then write a characterizing of our needs that include the artifacts we are going to develop, the stakeholders and the requirements to the artifacts. This can be found in Chapter 4.
Evaluate an existing tool for our purpose	The second phase is to have a case study of an existing tool, and from that see which functions the tool has. We evaluate it with respect to the requirements we defined in the first phase. The evaluation is found in Chapter 6. After this process we make a description of how we are going to develop the artifacts, and redefine our requirements based on the evaluation. The redefined requirements are described in Chapter 4.
Develop the artifacts	From the description of functions and the requirements we redefined in the second phase we start making our method, and develop tool support for it. This will be an iterative phase, where we evaluate our functions while developing one after another, and then improve them if necessary. We use the MAPS-breakdown step to split the method into different part. The evaluation of each part will be based on the redefined requirements from phase two. A description of the developed artifacts are found in Chapter 7.
Evaluate the artifacts with respect to our needs	The last phase will begin after we have developed our method, and its tool support. Here we take an evaluation of the method, and see how well it fulfills its requirements. We also evaluate the tool. From this we see how well we did, and what can be done next. In this phase we do not go back and improve the artifact, but rather log the result of the evaluation, and see what can be improved in a future version. The evaluation of our artifacts are found in Chapter 8.

Table 3.1: Development process of this project

3.2 Evaluate an Existing Tool for our Purpose

One way of getting requirements to new tools are to evaluate existing ones [40]. Therefore, in order to get a better overview of how our method should work, and how the tool should be, we take a look at an existing tool.

We conduct the evaluation with a case study, and then refine our requirements. We use the MAPS' specification step, and make tests based on input and desired output. We make the tests before we start the evaluation, and then when evaluating see how well the existing tool solves the task, and gives the desired output. Since the existing tool is not made for testing policy adherence, we try to adapt the tests to see how much of the policy adherence test the existing tool can perform.

3.3 Develop the Artifacts

This phase will be an iterative phase. We start by develop our method, based on the result from the evaluation of the existing tool and our defined requirements. We concentrate on one part of the method at the time. We then develop tool support for one part at the time. When tool support for each of the parts is developed we evaluate it based on the desired result and the requirements defined.

We continue to develop more parts of the method, evaluate it, and improve if necessary, until all parts is developed, evaluated, and improved.

3.4 Evaluate the Artifacts with Respect to our Needs

We start with evaluating the method. This will be done by conducting the same kind of case study as we did with the existing tool, only more detailed. We want to see if it concludes with the desired result when we test some specifications. With this evaluation we will be able to evaluate all the requirements, both functional and non-functional.

We also want to evaluate our tool. We conduct the evaluation the same way as for the method. We give our tool some input, and see if we get the desired result as output. We will also have a look at the time the tool spends solving the different test cases. With this approach we can evaluate the functional requirements to the tool. In order to evaluate how well our tool supports the non-functional requirements, we will need to have a different evaluation [50].

Some of the non-functional requirements to our tool we can evaluate while evaluating the functional requirements of the tool, however with another point of view. The other

non-functional requirements have to be evaluated with the use of analyzing the tool, and see how it actually works, and how the user will experience it.

Chapter 4

Characterizing the Needs

Before we can start developing the artifacts, there are some issues we have to discuss. First we find our stakeholders, and define their role with respect to our artifacts. Then we go through the requirements of the artifacts.

4.1 Stakeholders

There are a lot of different people that can interact, or get affected by the artifacts we want to develop. We have to define our stakeholders early in this process, in order to have their opinions in mind while planning the development process [40] [29].

The different stakeholders will have different point of views of what is important and what is not important for the functionality of the artifacts. They may also have opinions on how they should look, and be used. We therefore need to look at all the different point of views while defining the requirements.

Our primary stakeholder is our end-users, who will use our method when they develop, and test new system specifications. We want to create a method, with tool support, for them to use. The method will help them with testing the system specification they are developing, in relation to some policy specifications they have defined. Our end-users will be system-developers, testers, and the ones that will make the policy specifications. It is not certain that there is an equally clear distinction between these roles in all projects, but we choose to split them such a way.

When defined our primary stakeholders, we also need to find out if they will include more people to interact with the artifacts, for example their end-users. This version of the method will not have all the functionality that would be required for being the main method used in development, so we will not take into account any other stakeholders than

the ones mentioned above. In future versions, where our stakeholders take their clients as participants in the testing process, this of course has to be taken into account. The stakeholders we will take into account are therefore the following.

- Tester

  One of our primary stakeholders is the ones that will conduct the testing by using our method. They will get a system specification and some policy specifications, and use our method to test if the system specification adheres to the policy specifications.

- System-developers

  The system-developers will use our tool while defining their system specification. Our tool will also be able to work with specifications defined in other UML-tools, but if the system-developers do so, they have to make the specification in such a manner that the tester can do the testing using our tool without to much work in converting the specification.

- Policy-developers

  The policy-developers will have the same interaction with the tool as the system-developers. They can define the policy specifications using our tool.

4.2 The Artifacts and their Requirements

When defined our stakeholders, we begin to find the requirements to our artifacts. We can categorize requirements into several different types [52] [40]. We have only focus on two types; functional requirements, and non-functional requirements.

- Functional requirements

  The necessary task, action or activity that must be accomplished, or what the system or one of its components must do [31].

- Non-functional requirements

  These requirements are more about how easy the product is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions occurs [46].

We will in this project produce two different artifacts, which are listed below.

- A method for testing policy adherence

- A tool for supporting the method

  The method for testing policy adherence will be supported by the tool.

  Below we have listed some parts the tool will consist of. The tool might not be physically like this, but the purpose is to describe the main features of the tool.

  - An editor

    The editor has the role of providing the functions to create, modify and delete sequence diagrams, and opening a text document.

  - A test-generator

    The test-generator has the role of the actual policy adherence testing. It has to be able to test policy adherence between a system specification and a policy specification.

  - A report-generator

    The report-generator has the role of making a report with the result from the test-generator.

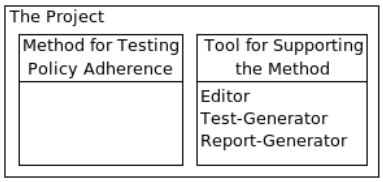


Figure 4.1: The artifacts of this project

The artifacts of the project are illustrated in Figure 4.1. For a more detailed description of the developed artifacts consult Chapter 7.

For the different artifacts in the project we made different requirements. The different artifacts are used in different ways, so we looked at what is important for each of them. The different stakeholders will have different opinions on which requirements that are important for which artifact. They may only have opinions towards the artifacts they will interact with, or the ones they know. Since we know all the artifacts and their functionality it is our job to point out the requirements, but with the stakeholders in mind.

In the next sections we list and describe the requirements for the different artifacts. We list the requirements to the method first, and then the requirements to the tool. We also list the requirements to the different parts of the tool in Section 4.2.2.1- 4.2.2.3.

In Chapter 8 we evaluate the developed artifacts with the respect to these requirements.

4.2.1 Requirements for the Method

We start with listing the requirements to the method.

Functional Requirements

Requirement 1 *support testing policy adherence between a system specification and a policy specification*

Requirement 2 *facilitate the policy adherence test by generating test diagrams based on a system specification and a policy specification*

The method should facilitate that the testing can be done by generating test diagrams based on a system specification and a policy specification. Those test diagrams and the system specification should then later be tested against each other.

Non-Functional Requirements

Requirement 3 *be a well defined method*

For the method to be used, it is important that it is well defined. The method has to be defined in a way so it can be used, and improved by others. With well defined, we mean that the method should describe a "systematic way of accomplishing something" [32], and *something* here being testing policy adherence. The stakeholders that are supposed to use this method, we assume have experience with both modeling and testing. Each step in the method needs to be defined with desired input, provided output, and a description of how to conduct the step, and this in a manner the stakeholders will understand.

4.2.2 Requirements for the Tool

Below we list the requirements for the tool. We also list the requirements for the different parts, or roles the tool will consist of in Section 4.2.2.1- 4.2.2.3.

Functional Requirements

Requirement 4 *support the method for testing policy adherence*

The tool is being made for supporting our method for testing policy adherence, and has to conclude with the same result as the method. It also has to conduct the same steps as the method conduct.

Non-Functional Requirements

Requirement 5 *be user-friendly*

In order for the tool to be used it has to be user-friendly. The functions have to be easy to perform, and the steps the user needs to perform have to be intuitive.

Requirement 6 *be efficient*

The tool has to do its task in a time-interval that is realistic with respect to the size of the specifications it gets as input. It can not spend more time than what a person is willing to wait for. The persons using the tool want to have the answer back in a time-interval he thinks is realistic in relation to the size of the tests he runs. If the test is small, it might be seconds, or if the test is very big, maybe minutes.

The main thing we have to focus on here is that the different parts in the tool sends their data as efficient as possible to the other parts, and that each part does their task as efficient as possible. In order for the tool to be efficient, the method also has to work in an efficient way.

4.2.2.1 Requirements for the Editor

The editor has the task of providing the functionality for the user to create, modify and delete sequence diagrams. There will be two types of sequence diagrams; policy diagrams and system diagrams. The editor also has to support the viewing of the report from the testing. Below we have listed its functional requirements.

Requirement 7 *support creating, modifying and deleting of sequence diagrams describing a system specification*

The user has to be able to make system specifications as sequence diagrams.

Requirement 8 *support creating, modifying and deleting of sequence diagrams describing a policy specification*

The user has to be able to make policy specifications as sequence diagrams.

Requirement 9 *support the viewing of a read-only text document*

The document for the report will be a *read-only* text document. It will be made *read-only* to prevent it from being changed.

4.2.2.2 Requirements for the Test-Generator

The test-generator has the main task; the testing. It will have to generate test diagrams, test the policy adherence, and conclude with some results that can be passed on to the report-generator. Below we have its functional requirement.

Requirement 10 *conclude on an answer that answers the policy adherence question*

What the test-generator should do is to test if there is adherence between the system specification and policy specifications. It will get a system specification and a policy specification as input, test them, and then send out some results. The results can be that the system specification adheres to the policy specification, that it does not adhere, or that no result was found. The results it generates have to answer the question, or give an answer on why it did not find an answer.

4.2.2.3 Requirements for the Report-Generator

We need the results in the report to be useful to the users. Our end-users have to be able to see at the first glimpse on the report if the adherence is satisfied or not. If it is not, the results have to give good enough information of what is wrong, and where. Below we have listed its non-functional requirements.

Requirement 11 *be logical*

The information in the report must be organized in a way so the most important information comes first, and that it contains just the relevant information.

Requirement 12 *contain all, and only, the information needed*

All the information in the report has to be of relevance for the problem the tool should solve. Therefore the report should only contain information about what where tested, what results where concluded on, and what the results where based on. It has to contain enough information, without containing too much.

Chapter 5

State of the Art

In this chapter we look at work other people have done on the subjects of this project. In Section 5.1 we have a look at modeling in general, and on different modeling languages, including UML, a language that is reckoned as the de facto standard for modeling software applications [55]. In this section we also have a better look at STAIRS, which is made based on a subset of UML.

In Section 5.2 we continue with an extension of STAIRS named Deontic STAIRS, which can express policy specifications. Since one of the main topics in this project is testing, in Section 5.3 we have a look at existing methods that conduct testing with the use of sequence diagrams. In Section 5.4 we have a look at different tools for making sequence diagrams, and to do testing based on those sequence diagrams. In particular we take a closer look at a tool named Escalator, which provides an implementation of STAIRS.

5.1 Modeling Language

Modeling is a broad term, and there exist a lot of different models, all depending on what we want to express. We can use models for example when building a house, representing data, and when building a software system. The one common feature between all models is that they are used to make a description of something. We have focus on software modeling in this section.

Not everyone see the same use of modeling in software development [13], but there are clear advantages when modeling a software system before we start implementing it. To mention some of the benefits: "enhanced communication, better planning, reduced risk, and reduced costs" [13]. There are many different aspects that can be modeled when developing software. It therefore also exist many different models, all with their purpose, and many

different modeling languages to use for different purposes.

A modeling language is a language for making models. It has both a syntax and semantics. However, how formal or well-defined the syntax and semantics are will vary. Most modeling languages also have both a graphical and textual syntax.

In this project we have a particular focus on sequence diagrams. We therefore go through two modeling languages that support these kinds of diagrams in the next two subsections. First UML, which as mentioned is the most used modeling language, and then STAIRS, which is based on a subset of UML.

We need a modeling language that has a graphical syntax we can use to draw sequence diagrams, since we want our tool to support a language that gives the user the option to draw sequence diagrams. The sequence diagrams also need to be represented in a format, for example textually, which can be processed by our tool when analyzing what the sequence diagram contains. The format has to contain the information the tool needs, and be as easy as possible to decrease the work of the tool while analyzing the sequence diagrams. We therefore have a focus on how the two languages represent the sequence diagrams textually.

5.1.1 UML

Here we look at the most used software modeling language, Unified Modeling Language 2.0(UML 2.0) [55] [9] [4]. With UML we can make different diagrams that can be used for different purposes in many areas and domains. Most commonly UML is used for software development. All the different diagrams we can make using UML are listed below with a description on what they contain. The list and description are found in [4].

- Class Diagram

  Shows a set of classes, interfaces, and collaborations and their relationships.

- Object Diagram

  Shows a set of objects and their relationships.

- Component Diagram and Composite structure Diagram

  Shows an encapsulated class and its interfaces, ports, and internal structure.

- Use Case Diagram

  Shows a set of use cases and actors and their relationships.

- Sequence Diagram

Shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. This diagram also emphasizes the time-ordering of messages.

- Communication Diagram

  Shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. This diagram also emphasizes the structural organization of the objects or roles that send and receive messages.

- State Diagram

  Shows a state machine, consisting of states, transitions, events, and activities.

- Activity Diagram

  Shows the structure of a process or other computation as the flow of control and data from step to step within the computation.

- Deployment Diagram

  Shows the configuration of run-time processing nodes and the components that live on them.

- Package Diagram

  Shows the decomposition of the model itself into organization units and their dependencies.

- Timing Diagram

  Is an interaction diagram, like sequence diagrams and communication diagrams. This diagram shows actual times across different objects or roles.

- Interaction Overview Diagram

  Is a hybrid of an activity diagram and a sequence diagram.

Since we want our tool to work with sequence diagrams, we have a closer look at UML sequence diagrams. UML have several operators that are useful when defining a sequence diagram that represent a system specification [9]. We have listed the operators below, with a description from [55]. In Appendix A we have illustrated the different operators, with the use of sequence diagrams. The term *interaction fragment* used in the descriptions is a term used on the boxes illustrated in Figure 5.1 on page 26. The figure illustrate the interaction fragments of a neg-operator and an alt-operator.

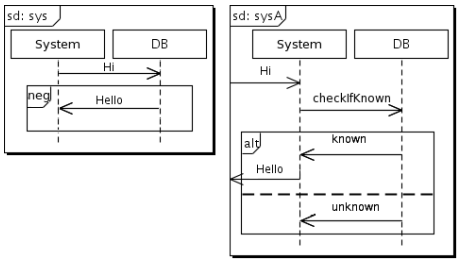


Figure 5.1: Two sequence diagrams with interaction fragments

- opt

  Indicate that the interaction fragment's condition has to be true for the events inside of the fragment to be executed.

- break

  Indicate that the interaction fragment should execute, and then terminate.

- loop

  Indicate that the interaction fragment will execute until the condition is false. There are three types of loops, and they all are different in the syntax of the condition.

- critical

  Indicate that the set of events in the interaction fragment must be treated as an atomic block.

- neg

  Make a set of events that are considered invalid. This operator makes the traces containing the events in the interaction fragment negative traces. It also has the option that the execution can skip the interaction fragment, and will then make positive traces that are equal the negative ones, with the exception that they do not contain any of the traces from the neg-interaction fragment.

- assert

  Indicates that the events in the interaction fragment are the only valid execution path. All other traces will be negative.

- ignore

  Specify that the messages it has as parameter can be safely ignored.

- consider

  Specify that the messages it has as parameter are explicitly relevant, and that all other messages safely can be ignored.

- seq

  Means *weak sequencing*, and indicate that the events occurrences in each operand can be interleaved according to these rules:

  – The ordering of the event occurrences within each operand is maintained.
  – If event occurrences in different operands occur on different lifelines, they can be interleaved in any order.
  – If event occurrences in different operands occur on the same lifeline, they can be interleaved only in such a way that the event occurrences of the first operand execute before the occurrences of the second operand.

- alt

  Is used to have alternatives to future designing. When implementing the system specification illustrated in the sequence diagram, a choice of what alternative to be implemented must be made.

- par

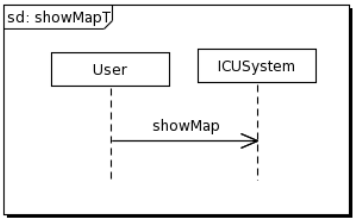
  Indicates that the interaction fragment may be merged and executed in parallel.

- strict

  Indicates that the ordering of the event occurrences is significant across lifelines, not just within the same lifeline.

We also want to know how UML represent a sequence diagram textually. UML saves the models in the XML Metadata Interchange (XMI) format [47]. XML is short for Extensible Markup Language. XML is designed for transporting and storing data, and are being used for many purposes like; web sited, documents, databases, and models. It is also meant to be both human-readable and machine-readable. XMI is a way to save UML models in XML, and provide the ability to move UML models between tools [47].

In Figure 5.2 on page 28 we have an example of a sequence diagram in UML, both textual in XMI-format, and graphical. The sequence diagram represents a system, ICUSystem, which receives a message, showMap, from a user, User.




```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML" xmi:id="_d3KxwGnoEeGaPKOjNJUKyA" name="model">
  <packagedElement xmi:type="uml:Interaction" xmi:id="_d3Mm8GnoEeGaPKOjNJUKyA" name="pol2B">
    <lifeline xmi:id="_lj07cGnoEeGaPKOjNJUKyA" name="User" coveredBy="_oWwWEWnoEeGaPKOjNJUKyA"/>
    <lifeline xmi:id="_mNUkYGnoEeGaPKOjNJUKyA" name="ICUSystem" coveredBy="_oWxkMGnoEeGaPKOjNJUKyA"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_oWwWEWnoEeGaPKOjNJUKyA"
name="Message0Send0" covered="_lj07cGnoEeGaPKOjNJUKyA" event="_oWwWEGnoEeGaPKOjNJUKyA"
message="_oWvvAGnoEeGaPKOjNJUKyA"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_oWxkMGnoEeGaPKOjNJUKyA"
name="Message0Recv0" covered="_mNUkYGnoEeGaPKOjNJUKyA" event="_oWw9IGnoEeGaPKOjNJUKyA"
message="_oWvvAGnoEeGaPKOjNJUKyA"/>
    <message xmi:id="_oWvvAGnoEeGaPKOjNJUKyA" name="showMap" messageSort="asynchCall"
receiveEvent="_oWxkMGnoEeGaPKOjNJUKyA" sendEvent="_oWwWEWnoEeGaPKOjNJUKyA"/>
  </packagedElement>
  <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="_oWwWEGnoEeGaPKOjNJUKyA"
name="SendOperationEvent0"/>
  <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="_oWw9IGnoEeGaPKOjNJUKyA"
name="ReceiveOperationEvent0"/>
</uml:Model>
```

Figure 5.2: The graphical and textual syntax of UML

We see that XMI has a lot of information about the sequence diagram, and that is one of the drawbacks with XMI. Since it can be used to describe all objects of UML, it has a complex syntax.

Our tool has to know what lifelines that are present in the sequence diagram, what messages each lifeline send and receives, and in which order they are sent and received. We get this from XMI, but also much more.

5.1.2 STAIRS

In this section we have a look at STAIRS [35] [27] [17] [18], which is a formal approach to system development with UML sequence diagrams.

STAIRS is based on a subset of UML, and below we have listed the operators STAIRS have common with UML.

- seq
- par
- strict

- opt
- neg
- assert
- loop
- alt

The operators above have the same meaning in STAIRS as in UML, so to see the description of them consult Section 5.1.1.

STAIRS also have some additional operators we have listed below with descriptions from [27].

- refuse

  Has the same meaning as neg, but with refuse we do not have the skip-option.

- xalt

  xalt might look like the alt-operator, but differs. When using xalt we specify that all alternatives in the interaction fragment have to be a choice in the implementation.

- skip

  Represents an empty sequence diagram.

The operators UML have that are not included in STAIRS are:

- break
- critical
- ignore
- consider

When not having the break-operator we can not easily specify that the fragment interaction should terminate after being executed. This will be the same as not having break in the programming language Java [6]. We do not lose functionality by not having break, we would just find another way of saying that the fragment interaction should terminate.

The operators ignore and consider are operators that allow some messages to be ignored. This might be a helpful functionality, but are not required when defining a system specification. So we do lose functionality with not having these operators, but there will not set limits for defining a system specification.

The last one, critical, indicates that an interaction fragment must be treated as an atomic block. This one will only have a function inside of a par-operator. If modeling a system specification that has a lot of parallel processes, STAIRS would not be a good idea to use, but there are ways to solve the atomic-problem. The easiest solution would be to have less parallel processes.

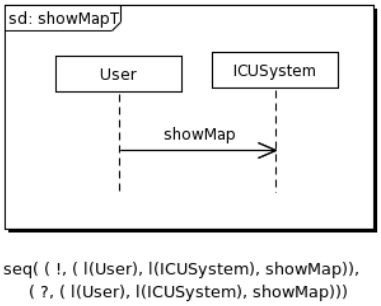


Figure 5.3: The graphical and textual syntax of STAIRS

In Figure 5.3 we have illustrated the two different syntaxes STAIRS have. The above one shows the graphical syntax, and the below one shows the textual syntax. If we look at the textual syntax it is relatively less complex than XMI.

In this project we will use this language to represent system specifications, both graphically and textually.

5.2 Policy Specification Language

STAIRS does not have the operators to define a policy specification. The policy specifications have to be defined different from the system specifications in order for the tool to distinguish between the system specification and the rules for the system specification.

There exist different ways and languages for represent a policy specification. Some of the different languages are *Traffic flow policy languages* [5], *Policy Description Language (PDL)* [26], and *OASIS eXtensible Access Control Markup Language* [5]. We only have a look at one language, Deontic STAIRS, which is based on STAIRS.

5.2.1 Deontic STAIRS

Deontic STAIRS [45] is a policy specification language, containing all the operators STAIRS have, but also operators to define a policy specification. The operators with which Deontic STAIRS extends STAIRS are the following.

- rule

  Is not exactly an operator, but indicates that the sequence diagram contains a policy specification.

- trigger

  Indicates that the interaction fragment is a trigger-interaction fragment.

- obligation, prohibition, and permission

  Indicates the type of the policy specification, and that the interaction fragment is a body-interaction fragment.

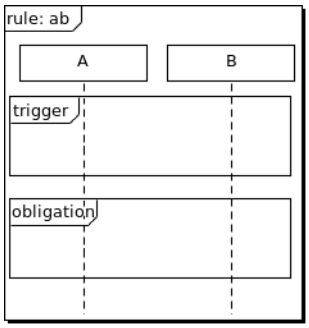


Figure 5.4: The graphical syntax of a policy specification

Figure 5.4 shows how a policy specification looks like in Deontic STAIRS. We see it has a name rule, and contain two parts. The first part is the trigger with the name trigger, and the second part is the body with the name obligation. The name of the body can be either obligation, permission, or prohibition, giving the body different meaning. The trigger contains messages, which make traces that will trigger the body to apply. When the traces in the trigger are executed in a system-trace, the traces in the body are required, allowed, or forbidden to be executed any given time after in the trace.

We will use this language to represent policy specifications in this project.

5.3 Model Based Testing

In Section 2.3 we gave an overview of testing techniques, and how to test sequence diagrams. In this section we look at different approaches that conduct testing with sequence diagrams.

There exist several methods based on generating test cases from UML sequence diagrams [12] [44] [38] [37]. They all have different approaches, and not all uses only sequence diagrams to make the test cases.

The method in [44] generates test cases based both on sequence diagrams and state diagrams. The state diagrams are used to define the state where the test cases defined in the sequence diagrams are to be triggered.

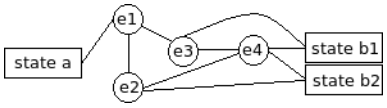


Figure 5.5: A graph based on a sequence diagram

In [38] the sequence diagrams are transformed into another format. This format is a graph that shows all system-runs that can go from a beginning state `a` to all ending states `b` that can be reached from `a` by one or more events `e`. We have illustrated an example in Figure 5.5.

The method then generates test cases from the graph, which defines input, output and post-conditions to the system specification. The method is implemented but do not cover executing of the cases. The method in [37] also generates test cases by making a graph from the sequence diagrams, but do not cover execution of the test cases.

IBM has a method for testing sequence diagrams. This method is made with the purpose to determine whether messages from two sequence diagrams are identical [22]. The method does not give a yes or no answer, but rather where and why they are different. The method work as following: it "take all events (message departures and arrivals) in order, and compare them without using the exact time. This kind of comparison, although simple, still shows when two sequence diagrams are essentially identical." [22].

From this we see that testing sequence diagrams against each other are not as widespread, but are an ongoing theme. In Section 5.4.1 we have a look at another method that is supported by a tool named Escalator. This method test two sequence diagrams based on generating test diagrams.

5.4 Tool Support

As mentioned in the beginning of Chapter 3, we want to have tool support for our method. Therefore we have a look at existing tools in this section, to see how others have solved testing with sequence diagrams, and how these approaches define their sequence diagrams.

There exist many tools to define sequence diagrams, some of them UML sequence diagrams, but also others with an informal syntax. We have listed some below. We use the term *combination fragment* below, which is an interaction fragment that defines a combination of interaction fragments [10].

- Sequence Diagram Editor [42]

  In this tool, it is possible to make lifelines, messages, and blocks. The blocks do not have the same appearance as combination fragments in UML, but can be an illustration for it.

  This tool saves the sequence diagrams in .sds-files, with the use of XML.

- Trace Modeler [25]

  This tool does not have that many operators, but is a tool we can use to model small sequence diagrams.

  The sequence diagrams are with this tool saved in .tmt-files with an informal syntax.

- Quick Sequence Diagram Editor [48]

  When making a sequence diagram in this tool, we write in the elements we want to add. It is possible to make all the operators UML have, but there might be that there is difficult making some of the operators.

  The sequence diagrams are saved in .sdx-files, with the use of XML.

- Altova UModel [1]

  Altova is a complex tool, which support a lot more models then sequence diagrams. But the sequence diagrams we can make with this tool do not support all UML-operators. The combination fragment it has is alt, loop, seq and ref.

  The sequence diagrams are saved with the use of XMI in this tool.

- Pacestar UML Diagrammer [43]

  When we open this tool it is like a text document, like Microsoft Word [8], where we can add different objects. There are lifelines, messages, and different combination

fragments that can be added. The name of the combination fragments are being added manually, so there are no restrictions on what type they can have.

The sequence diagrams are saved as .edg-files with an informal syntax.

- Papyrus [7]

  This tool can either be installed by itself, or as a plugin to Eclipse [11]. It also contains the most of the UML-operators.

  The sequence diagrams in this tool are saved as .uml-files in XMI-format.

There are also several tools online, where we can make sequence diagrams without installing the tool. To mention some:

- Gliffy [14]

  Here we can draw a sequence diagram in our web browser. This tool do not support that many operators, but have lifeline, messages, and activations.

  The sequence diagrams are not saved on the computer, so there are not possible to open the files containing the sequence diagrams if not using the program.

- Web Sequence Diagram [23]

  With this tool we do not draw the sequence diagram, but writes the elements we want to add. If we write User->System: start we get a diagram with a lifeline User, a lifeline System, and a message with the name start that are sent from User to System. This tool does not support all UML operators, but have both alt, opt, and loop. So this one is slightly more complex then Gliffy.

  The textual representation of the sequence diagrams with this tool is saved in the format we write, so relatively easy and human-readable.

In [12] they have made a tool, SeDiTeC, which uses sequence diagrams to generate test for Java programs [6]. Since this method do not test sequence diagrams against each other, we do not go deeper into the tool.

There are not that many tools that conduct testing with sequence diagrams, but there is one tool we know, and we go through it in the next section.

5.4.1 Escalator

In this section we have a look at a tool named the Escalator [27] [28], which is made as a plugin to Eclipse [11]. With this tool we can define sequence diagrams with STAIRS. This

can be done either with the textual syntax, using a text editor, or the graphical syntax, using a UML-tool. Then the tool can test if one system specification is a refinement of another. The overview of the functions the Escalator has is listed below. To get a more detailed description consult [27].

- Configure semantics
- Generate textual representation of sequence diagrams
- Generate tests
- Execute tests
- Refinement testing
- Generate traces
- Generate random traces
- Generate all traces
- Refinement verification

With this tool we can test refinement between two sequence diagrams. Refinement means we can go from an abstract high-level model, and then go down to a low-level model by refinement. On each level we go down, the specification are being strengthen. To get a more detailed description of refinement consult [36].

The tool has two different approaches to test refinement. One approach test refinement by generating test diagrams, the other one by generating the traces to the two sequence diagrams.

There are different types of refinement, but we only look at the most general one, general refinement. In order for a sequence diagram B to be a general refinement of a sequence diagram A:

- Every negative trace in A is also negative in B
- Every positive trace in A is either negative or positive in B

When Escalator tests refinement with the use of traces, all traces to both sequence diagrams are generated, and are then being compared.

We go through the second approach in Section 5.4.1.1 and 5.4.1.2. In Chapter 6 we go further into this tool.

5.4.1.1 Generate Tests

The second approach generates test diagrams that can be seen as test cases. The test diagrams will only contain one lifeline, and have messages from and to that lifeline. When testing, the test diagram will send messages to a system specification under test, and observe any messages it will get back. It will also be able to observe *nothing*, defined as theta. This *nothing* will be that the system specification under test does not send any message back. Based on the observation done by the test diagram, it will give a verdict, either pass, inconclusive, or fail for each trace.

The algorithm for generating test diagrams is described in [27] Section 13.

Example

Here we give an example. We have the system specification sysA in Figure 5.6, where we also show its traces. From the system specification we get two test diagrams generated, which are illustrated in Figure 5.7. We also show the traces of the test diagrams. The system specification represents an object, System, that first receives a Hi-message, then checks, by sending a message to the object DB, if the sender is known or unknown. If the sender is known the System sends a Hello-message back, and if it is unknown it does not send anything.

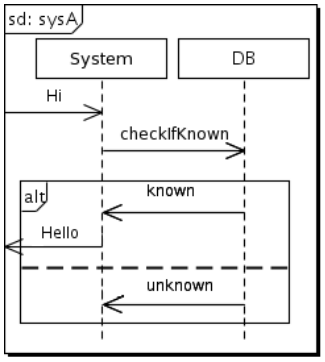


Figure 5.6: A system specification we generate test diagrams from

The messages in the test diagrams are the ones that go to or from the *outside* of the sequence diagram containing the system specification sysA. This is because the test diagram will test what the system specification in the sequence diagram, when given input, gives as output.

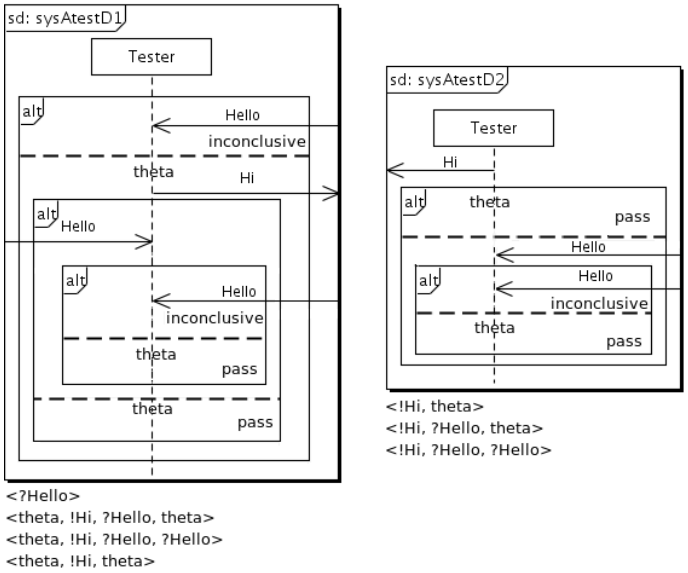


Figure 5.7: Test diagrams generated from the system specification sysA

If we look at the first test diagram in Figure 5.7, it starts by observing a system specification under test, let us call it sysB. It can either observe sysB sending a Hello-message, or theta. If there is a Hello-message the trace should have the verdict inconclusive, since the trace is not in sysA. If there on the other hand is theta , the test diagram observes theta, and then sends a message to sysB. There is only possible to send the Hi-message. After sending the Hi-message, the test diagram starts observing again. It is either Hello or theta that can be observed. If it is theta, the trace should have the verdict pass. This since the trace is present in sysA. If it is a Hello-message, the test diagram will see if it can observe more messages. So again it sees if it gets theta or a Hello-message. If it is a Hello-message, the trace is not in sysA, so the verdict should be inconclusive, if on the other hand it is theta the verdict should be pass.

The second test diagram in Figure 5.7 starts with sending a message Hi, and then starts observation. Either is can observe theta or Hello. sysA contains a trace that contains Hi, so the verdict should be pass. sysA also contains a trace containing Hi and Hello, so that trace also should have the verdict pass. All other traces should have the verdict inconclusive.

The Escalator executes this algorithm, and makes the test diagrams, which later can be tested against another specification.

5.4.1.2 Execute Tests

The test algorithm in the Escalator test some test diagrams against a specification.

The test begins with the test diagram interacting with the specification. It sends messages to the specification, and observes what the specification sends back. This happens like explained above.

When reaching a verdict in a test diagram, the Escalator does an analysis to find the final result. If the trace-verdict is pass, the final result will be pass. If the trace-verdict is inconclusive, the final result will also be pass. If the trace-verdict is fail the final result can be either pass or fail. It will be fail if the test diagram does not contain more negative traces than the negative traces that are also in the sequence diagram tested against. In all other cases the final result will be pass.

What the Escalator does not have is support for Deontic STAIRS. It does not directly support testing policy specifications defined in Deontic STAIRS together with a system specification.

Chapter 6

Evaluation of the Escalator for our Purpose

In order for us to get a better grip on how our method should test policy adherence, we have made two different approaches. These approaches are presented in this chapter, and we also evaluate them with respect to our purposes. We moreover evaluate how well the Escalator provides support for these approaches.

The first is an approach where we have the assumption that there exists a tool that supports all the functionality needed. This approach will test policy adherence with 100 percent accuracy. In the second approach we show a method that will give the result on the policy adherence, without testing 100 percent.

We first go through the steps of each approach, and give an example of it where we use functionality from the Escalator. After that we conduct an evaluation of both the approach and the Escalator.

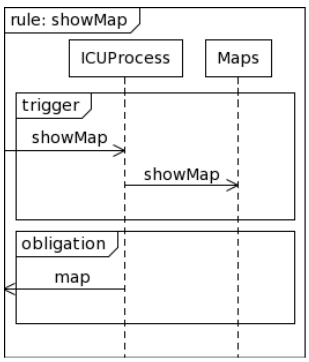


Figure 6.1: The policy specification used in this example

We have worked with an informatics student, I. Refsdal, and together defined a system specification, which can be found in Appendix B. The system specification is based on a system presented in a course at the University of Oslo, [21]. We also defined a policy specification illustrated in Figure 6.1 on page 39. In the examples of the approaches we will use these specifications.

As explained in Section 2.3.2 we only test the policy rule obligation, and prohibition. We do not include the rule permission.

6.1 Evaluation of the Escalator Automatic Refinement Checking

We go through the first approach here where we assume that we have a tool that supports all the functionality we need.

6.1.1 Step-Wise Process

This approach has three steps that will be described below.

1. Define a system specification and a policy specification

   The system specification is defined in STAIRS, and the policy specification is defined in Deontic STAIRS. The output from this step is the system specification and the policy specification defined in the tool.

2. Generate traces

   We then use a tool to generate the traces of the system specification and the policy specification. The output from this step is one set of traces for each specification.

3. Check adherence

   This step is divided into five sub-steps:

   (a) Get and keep the information of the relation between the system specification and the policy specification

   We need to have the information of which specification is the system specification, and which the policy specification. We also need to have the information of what type of policy it is.

   (b) Calculate the positive traces in the system specification by taking the positive traces, P, minus the negative traces, N

We are only interested in the traces that are positive, and not negative. So this step removes the negative traces and the positive traces that are also negative. We get as output all the traces from the system specification that is only positive.

(c) For all positive traces in the system specification, S, test if the trigger, t, is a sub-trace

Test if one trace from the trigger is a sub-trace of one of the positive traces in the system specification. If it is not in any trace, the system specification adheres to the policy specification, and the testing with the policy specification is done. If it is, we take the trace with us to the next step.

(d) Test the positive traces from the previous step, S', against the traces in the trigger, t, and body, b

If one of the traces of the trigger is a sub-trace of one of the traces of the system specification, we have to test if one of the traces of the trigger and body together is a subset of the traces of the system specification.

(e) Analyze the result

Then we have to see what type of rule the policy specification is. If it is an obligation the body is required to be present, if a prohibition the body is forbidden to be present.

After doing these steps we know for sure the adherence between the system specification and the policy specification.

In the next section, we go through an example, and look at the functionality of the Escalator. We see if we can use the Escalator for our testing, and if not, how it would have to be modified.

6.1.2 Example

Since the Escalator does not support all the functions we want, some of the steps will be done by the Escalator and some manually by a human.

1. Define a system specification and a policy specification

   In this example we have the system specification, ICU-system, defined in Appendix B, and the policy specification defined in Figure 6.1 on page 39. Since this is only an example we do not take the whole ICU-system. In Figure 6.2 on page 42 we have illustrated the part of the system specification we will work with here.

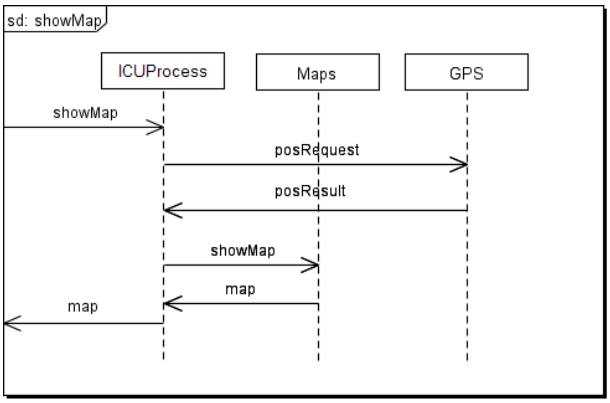


Figure 6.2: The part of the ICU-system used in this example

The policy specification we use in this example states that if a User ask the ICUSystem to see the map (with the message showMap), the ICUSystem should show him the map (with the message map).

The Escalator supports STAIRS, so it will be able to take the system specification as input. Since the Escalator does not support Deontic STAIRS, we have to split the sequence diagram containing the policy specification. In Figure 6.3 we have illustrated the split. The sequence diagram named showMaptrigger contains the trigger of the policy specification. The other sequence diagram, named showMap, contains both the trigger and body of the policy specification. This because we first have to test if a trace from the trigger is present in a trace from the system. If there are one or more traces from the system that contains a trace from the trigger, we also need to test if those traces contain both the trigger and body.

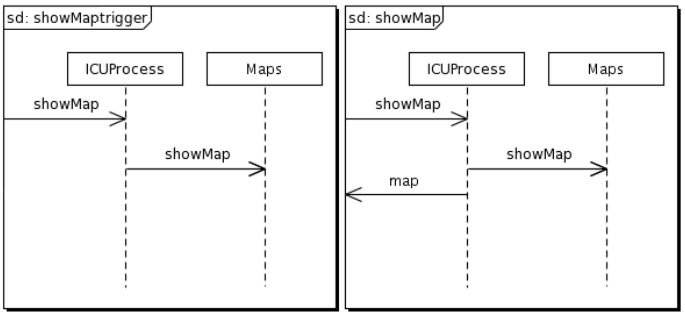


Figure 6.3: How the policy specification is changed to serve as input to the Escalator

2. Generate traces

   The Escalator has the functionality to generate traces from a sequence diagram, so we can use it in this step. After this step we have three sets of traces. Since this is a very small example, the three sets only contain one positive trace each.

   The positive trace of the system specification looks like this:
   <?showMap, !posRequest, ?posRequest, !posResult, ?posResult, !showMap, ?showMap, !map, ?map, !map>
   The positive traces of the trigger specification looks like this:
   <?showMap, !showMap, ?showMap>
   And, finally, the positive trace of the policy specification looks like this:
   <?showMap, !showMap, ?showMap, !map>

   Where, as mentioned in Section 2.3.2, the symbol ! represents the sending of a message, and ? represents the receiving of a message. The system specification will be defined by the use of one sequence diagram, but every policy specification will use two sequence diagrams. It is our responsibility to remember what trace that corresponds to what sequence diagram.

3. Check adherence

   The first step here is to collect the information about the relation between the three sequence diagrams. The next is to find the positive traces of the system specification. Since we only have one positive trace from our system specification, we do not need to do anything here.

   We then check if the trace from the trigger specification is a sub-trace of the trace from the system specification, and in this case it is. We have the system specification's trace here, with the trigger specification's events in *italic* font: <*?showMap*, !posRequest, ?posRequest, !posResult, ?posResult, *!showMap*, *?showMap*, !map, ?map, !map>. Then we check if the trace from the trigger and the trace from the body together are a sub-trace of the trace from the system specification, and that is also the case there. We can see it on the trace from the system specification here, with the trigger specification's event in *italic* font and the body specification's events in **bold** font: <*?showMap*, !posRequest, ?posRequest, !posResult, ?posResult, *!showMap*, *?showMap*, !map, ?map, **!map**>. Then we know the trace in the policy specification can happen in the system specification. Next we check if the policy specification has the type obligation or prohibition. This is an obligation rule, so the system specification adheres to the policy specification in this example. If it had been a prohibition the system specification

would not adhere to the policy specification.

6.1.3 Evaluation

From the simple example above we see there is some functionality that is missing in the Escalator for us to conduct the testing we want in this approach. The Escalator supports Step 1 and Step 2 if we split the policy specification in two. In order for this to be done in a tool without splitting the policy specification, the tool has to support the language Deontic STAIRS.

The sub-steps in Step 3 we have to do manually. To find the set of positive traces we need to look at the generated traces in the Escalator. The next two sub-steps involve more work. We have to go through the traces from the system specification looking for a trace matching the trigger specification, and if one trace exists, also look for a trace matching the body specification.

This steps works, and will give us the result we want. But there is one big problem; it can be very time consuming. The bigger the specifications are, the more traces they can contain, the longer time it will take to generate the set of traces, and the longer a human will use to search in the traces.

The functionality the Escalator is missing in order to do the steps defined above in Section 6.1.1 is the following.

- Support the language of Deontic STAIRS
- Capture the information of what sequence diagram that contains the system specification, and what sequence diagram that contains the policy specification
- Capture the information of what in a policy specification that is the trigger, and what is the body
- Capture the information of what type of rule the policy specification has
- Calculate the positive traces of a system specification that are not also negative
- Check if one of the traces from a trigger specification is a sub-trace of a trace from a system specification
- Check if one of the traces from a policy specification is a sub-trace of a trace from a system specification
- Write the result in a document

This functionality is not supported by the Escalator and we have to implement it in our tool.

In the example above we ignored the testing-functionality the Escalator [27] [28] has. It has the functionality to test refinement between systems. It has an algorithm for testing refinement that makes test diagrams from a sequence diagram and then test those test diagrams against another sequence diagram. In the next section we look at this.

6.2 Evaluation of the Escalator Interactive Refinement Testing

Here we go through the second approach. This approach generates test diagrams, and tests them against another sequence diagram. We use the testing to check if a trace from the trigger is present in a trace from the system. If it is, then test if a trace from the trigger and body together is present in the trace from the system. But since finding all the possible traces of a big sequence diagram can take too long time, we want to see how we can test *good enough* without testing all.

6.2.1 Step-Wise Process

This approach has three steps that will be described below.

1. Define a system specification and a policy specification

   This step is the same as the first step in the previous approach. Here the system specification is defined with STAIRS, and the policy specification with Deontic STAIRS. The output here is the system specification and the policy specification defined in the tool.

2. Generate test diagrams

   We have to choose which lifeline that should be tested. The lifeline needs to have the same name in both the system specification and policy specification. The tool will then analyze the lifeline in the system specification, and find all the messages that go to, and from it. The messages that go to the lifeline will be part of the system specification's input-alphabet, and the ones that go from the lifeline will be part of the system specification's output-alphabet. The same will be done with the policy specification. We then have two sets of input-alphabets, and two sets of output-alphabets. The information of what messages in the policy specification's alphabets that are from the trigger and from the body also need to be kept track of.

All the test diagrams will have the same output-alphabet, namely the union [24] of the policy specification's output-alphabet and the system specification's output-alphabet. The input-alphabet of the test diagrams will differ. The first test diagram's input-alphabet will be equal to the input-alphabet to the policy specification. But if the policy specification does not have any messages in the input-alphabet, it will take one of the messages from the system specification's input-alphabet. The more test diagrams that are being made, the more the input-alphabet grows by adding messages from the system specification's input-alphabet.

There will be made two sets of test diagrams. One set for testing if a trace from the trigger is present, and another set for testing if a trace from both the trigger and the body is present. The test diagrams testing if a trace from the trigger is present, needs to have the trigger-messages in the order they are in the policy specification. But it can have the messages from the body and system specification before, in between and after. The test diagrams testing if a trace from both the trigger and body are present need to have all the messages from the policy specification in the same order as they are in the policy specification. But other messages, from the system specification, can be before, in between and after.

The output from this step will be two sets of test diagrams.

3. Test the test diagrams

   We then test the test diagrams against the system specification. First we need to test the system specification together with the test diagrams made to check if a trace from the trigger is present. The result from the test can be pass or fail.

   The pass result on a trace means that the trace in the test diagram is present in the system specification. The fail result means that the trace in the test diagram is not found in the system specification. Since the result fail can mean that the test diagram is simply not including enough messages from the system specification's input-alphabet, it may be best to conduct further tests with more test diagrams.

   If all traces from the test get fail as result, no trace from the trigger is present, and the system specification adheres to the policy specification. The testing is then done.

   If, however, one or more of the traces get pass, it means that one or more of the traces from the trigger is present, and we need to test if one or more of the traces from the whole policy specification is present. The result from testing the test diagrams made to check for the presence of traces from the trigger and body combined, will state if the system specification includes a trace containing the trace from both the trigger

Policy	Trigger present	Body present	Result
obligation	yes	no	fail
obligation	no	yes	pass
obligation	yes	yes	pass
obligation	no	no	pass
prohibition	yes	no	pass
prohibition	no	yes	pass
prohibition	yes	yes	fail
prohibition	no	no	pass

Table 6.1: The desired result

and body. If there is a trace that get pass it means that the trace contains both the trigger and body. A fail result means that it does not contain the trigger and body.

If the policy has the type obligation, we want to get pass on all the traces that gave pass in the previous test. Then the system specification adheres to the policy specification. If the policy has the type prohibition, we want fail on all the traces that gave pass in the previous test, because in that care the system specification adheres to the policy specification.

In Table 6.1 we show what final result we want after testing if the trigger and body is present in the system specification.

6.2.2 Example

We go through these steps, and try to conduct them with support from the Escalator. We see how the Escalator makes the test diagrams with the test-functionality it has, and how it does the testing of those test diagrams. We also see how it can be used for our purpose.

1. Define a system specification and a policy specification

   The system specification we will use in this example is illustrated in Figure 6.4 on page 48 and the policy specification is illustrated in Figure 6.5 on page 49. We have illustrated the policy specification with the use of three sequence diagrams. The sequence diagram to the left contains the original policy specification defined in Deontic STAIRS, and the two to the right are the sequence diagrams we can give to the Escalator. The upper one contains the trigger of the policy specification, and the lower one contains both the trigger and body. The Escalator, as mentioned, can only take specifications defined with STAIRS.

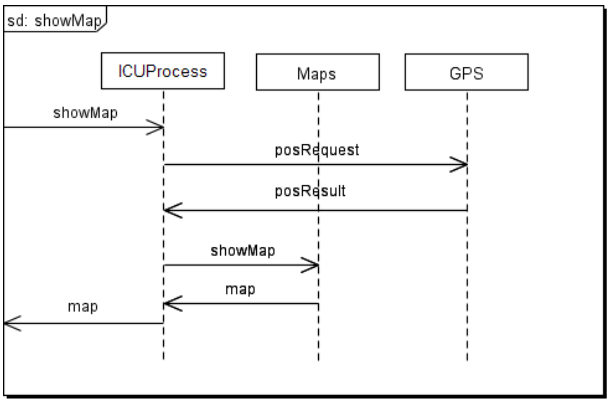


Figure 6.4: The part of the ICU-system used in this example

Diagram	**Input**	**Output**
System	showMap, posResult, map	posRequest, showMap, map
Policy	showMap	showMap, map

Table 6.2: The alphabets

2. Generate test diagrams

   In Table 6.2 we show the alphabet of the system specification and the alphabet of the policy specification. Before we could make them we had to choose a lifeline that would be tested. In this example we have chosen lifeline ICUProcess.

   The idea with the testing in the Escalator is that when we tell it to test a specification A against a specification B using *test down*, the Escalator makes test diagrams based on what is in specification A. These test diagrams are then tested against specification B. The Escalator also has *test up* that are not addressed here. The interested reader is referred to [27].

   In more details, if we give the Escalator the trigger specification from Figure 6.5 as input and tells it to generate test diagrams from it, we get the test diagrams illustrated in Figure 6.6 on page 50. As described in Section 5.4.1.1, we can observe *nothing*, which is denoted by theta in the test diagrams.

   These test diagrams are not represented as we need them for our purposes; they only contain the event that goes to and from the *outside* of the sequence diagram. We can therefore change the sequence diagram containing the policy specification to only contain the lifeline we will test, and will then get the test diagram illustrated in

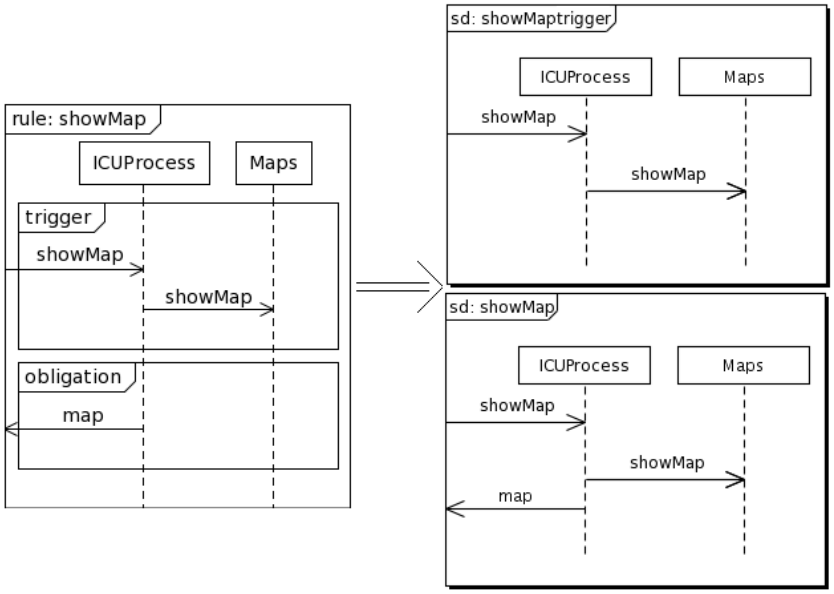


Figure 6.5: The policy specification used in this example

Test diagram	**Input-alphabet**	**Output-alphabet**
Test diagram 1	showMap	posRequest, showMap, map
Test diagram 2	showMap, posResult	posRequest, showMap, map
Test diagram 3	showMap, posResult, map	posRequest, showMap, map

Table 6.3: Making the test diagram's alphabet

Figure 6.7 on page 51 from the trigger specification. Figure 6.8 on page 54 illustrates the test diagrams generated from the whole policy specification.

The test diagrams are still not made as we want. This is because the Escalator generates the test diagrams based on only one specification, and we need to have them based on also the system specification. We will now see how the test diagrams have to be made.

We show the alphabets of the test diagrams as we would need them to be in Table 6.3. There will be three test diagrams because there are two messages from the system specification's input-alphabet that the test diagram can be extended with.

The first test diagram to check if the trigger is present is illustrated in Figure 6.9 on page 55, by showing the traces the test diagram would have. The reason why we do not show the sequence diagram is because of the size.

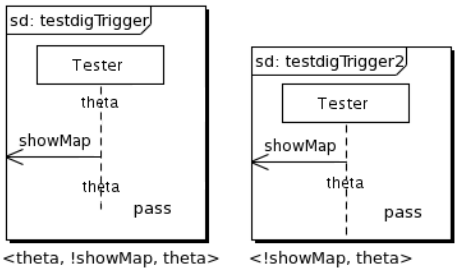


Figure 6.6: Two test diagrams generated in the Escalator from the sequence diagram containing the trigger

The second test diagram would contain 60 traces and the third 360 traces. Because of the size, we do not illustrate them here.

We also have the test diagrams to test the whole policy. The first test diagram is illustrated in Figure 6.10 on page 55 and the second one in Figure 6.11 on page 55, both by showing their traces.

The third test diagram to test the whole policy would have 120 traces, and is not illustrated here.

3. Test the test diagrams

   From the previous step we have the test diagrams the Escalator generated, and the ones we made manually.

   When we apply the test diagrams of Figure 6.7 made to test the trigger, which the Escalator generated, against the system specification illustrated in Figure 6.4 on page 48 we get the result:
   pass: <theta, !showMap, ?showMap>
   pass: <!showMap, ?showMap>
   We get pass, which should mean that the trigger is present. The trace that got pass does not contain all the events from the trigger, so this result does not answer what we want.

   We also check if the whole policy specification of Figure 6.8 on page 54 is present in the system specification of Figure 6.4 on page 48, by using the Escalator. The result after testing the whole policy specification is:
   pass: <theta, !showMap, ?showMap>
   pass: <!showMap, ?showMap>

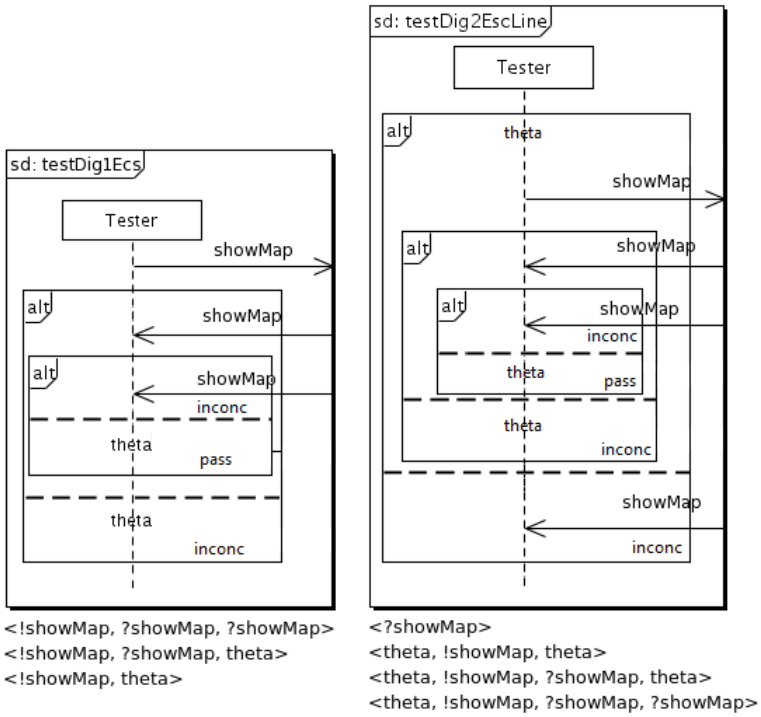


Figure 6.7: Two other test diagrams generated in the Escalator from the sequence diagram containing the trigger

Here we also get pass, but also this traces does not contain the events we would expect it to contain. When getting pass on a trace when testing if the whole policy specification is present in the system specification, we expect it to contain all the traces from the policy specification.

If we apply the test diagrams we made manually against the system specification with the test-functionality in the Escalator, we get different results. We get pass on empty traces, and some error, and fail-verdicts. The reason for these strange results is that we are trying to test something the Escalator does not know how to test. We are using the Escalator's function to test refinement while we want it to test policy adherence. The testing is also very time-consuming.

If we apply the first test diagrams we made manually for testing if the trigger is present, illustrated in Figure 6.9 on page 55, the way we want the new tool to do it, we get fail-verdicts on all traces. We will get fail since the traces are not including enough messages. The second test diagram will also result in only fail-verdicts, for the same reason. The third test diagram however, will return some pass-verdicts. This

is because some of the traces in the test diagram have the same events in the same order as in the trace from the system specification. Because of the pass-verdicts, we can conclude that the trigger is present.

Then we start to apply the test diagrams that check if the trigger and body together are present in the system specification. We want the first two test diagrams, illustrated in Figure 6.10 and Figure 6.11 on page 55, to result with fail-verdicts on all traces. This is for the same reason as for the first two test diagrams checking if the trigger is present, namely that they are not extended enough. The third test diagram will result with pass-verdicts on the same traces as when testing for the trigger. Those are the traces that have the same events in the same order as the trace of the system specification. From this testing we can conclude that the whole policy is present in the traces that contained the trigger. Since the policy has the type obligation, the system specification adheres to the policy specification.

6.2.3 Evaluation

The test-functionality in the Escalator uses an algorithm that takes a sequence diagram as input and produces test diagrams from it. It then tests those test diagrams against another sequence diagram. The result from a test can be pass or fail. The pass result means that the trace is present as a positive trace in the sequence diagram under test, and the fail result that it is present in a negative trace in the sequence diagram under test. The different results can also come at other scenarios, as it did when we tested the test diagrams we made manually. To read more about when the different verdicts apply, consult [27].

We want the test-algorithm to look at both the policy specification and system specification when making the test diagrams. This is one thing we have to implement for in our tool. The tool also has to support Deontic STAIRS so that we can give the policy specification as input defined in Deontic STAIRS. The tool also has to know what in the policy specification that is the trigger and what is the body, and what type of policy it is. There also has to be support for choosing a lifeline to test, without changing the sequence diagrams. This is things we have to implement in the new tool.

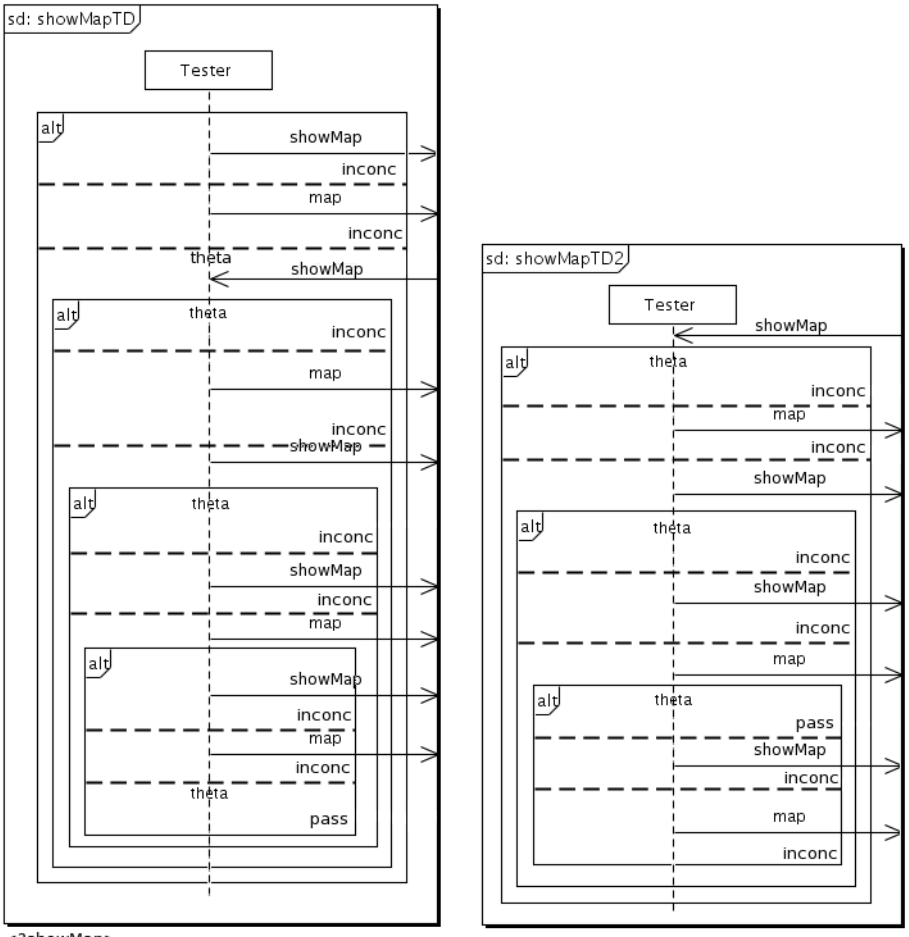
The test diagrams also have to have the messages in the right order. The test diagrams that check if the trigger is present must have the trigger-messages in the same order as they are in the policy specification. The test diagrams that check if both the trigger and body is present, have to have the trigger-, and body-messages in the same order as they are in the policy specification.

The verdicts the Escalator returns also need to be analyzed to finally determine the

answer to the policy adherence question.

The functionality the Escalator is missing in order to conduct the steps defined above in Section 6.2.1 is listed below.

- Support the language of Deontic STAIRS
- Capture the information of what sequence diagram that contains the system specification, and what sequence diagram that contains the policy specification
- Capture the information of what in a policy specification that is the trigger, and what is the body
- Capture the information of what type of rule the policy specification has
- Support the function of choosing the lifeline to test
- Generate test diagrams based on a system specification and a policy specification
- Analyze verdicts from the Escalator to answer the question of policy adherence
- Write the result in a document



<?showMap>
<?map>
<theta, !showMap, ?map>
<theta, !showMap, theta>
<theta, !showMap, ?showMap, theta>
<theta, !showMap, ?showMap, ?showMap>
<theta, !showMap, ?showMap, ?map, ?showMap>
<theta, !showMap, ?showMap, ?map, ?map>
<theta, !showMap, ?showMap, ?map, theta>

<!showMap, theta>
<!showMap, map>
<!showMap, ?showMap, theta>
<!showMap, ?showMap, ?showMap>
<!showMap, ?showMap, ?map, ?showMap>
<!showMap, ?showMap, ?map, ?map>
<!showMap, ?showMap, ?map, theta>

Figure 6.8: Two test diagrams generated in the Escalator from the sequence diagram containing both the trigger and the body

Figure 6.9: The traces of the first test diagram for testing the trigger

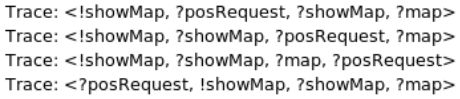


Figure 6.10: The traces of the first test diagram for testing the whole policy

Figure 6.11: The traces of the second test diagram for testing the whole policy

Chapter 7

Developed Artifacts

In this project we have aimed at developing a method for testing policy adherence, and making a tool for supporting the method. We wanted our method to test policy adherence by generating test diagrams, which later could be tested together with a system specification.

The test diagrams would be based on both a system specification and a policy specification. Since testing can be a time-consuming task, we also wanted our test diagrams to be generated in a manner that could save time, and if possible not be generated at all. The latter is because if the answer to the policy adherence question is found before the method generates test diagrams, the method should jump to writing out the result.

We wanted the first test diagram to only contain the input alphabet of the policy specification, but the output alphabet of both the system specification and policy specification. The test diagrams would then be successively extended by including more messages from the system specification's input alphabet.

From this, in the best case, we would find out how the system specification adheres to the policy specification by only testing the first small test diagram. The drawback with this method is that in the worst case the right answer of the policy adherence question is only found after we eventually have generated a test diagram containing all messages.

For this purpose we have developed two main artifacts:

- A method for testing policy adherence, named the Køller-method
- A tool supporting the Køller-method, named the Køller-tool

In this chapter we present the artifacts, and we start with Section 7.1 where we go through the Køller-method. The Køller-method consists of several steps that will be described in the order they are performed.

The Køller-method is supported by the Køller-tool. The Køller-tool uses the Escalator, and is made as a plugin to Eclipse [11]. When going through the Køller-method we also state how the different steps are supported by the Køller-tool.

In Section 7.2 we go more in details about the Køller-tool. We illustrate its structure, and describe its main components.

7.1 Køller-Method

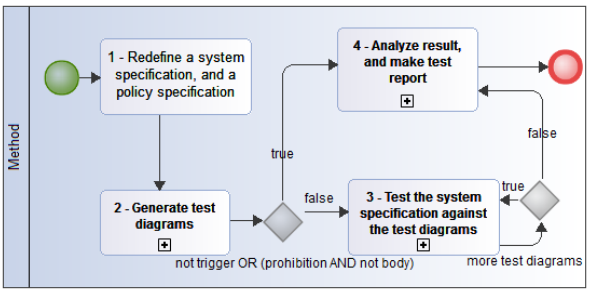


Figure 7.1: The steps of the Køller-method

The Køller-method is made for testing policy adherence. In order to achieve this task the Køller-method has four main steps:

1. Redefine a system specification, and a policy specification
2. Generate test diagrams
3. Test the system specification against the test diagrams
4. Analyze result, and make test report

We have illustrated the steps in Figure 7.1, and in the following we describe each step, state received input, provided output, and how the Køller-tool supports the steps.

7.1.1 Step 1: Redefine a system specification, and a policy specification

- **Objective:** Define the part of the system specification that needs to be tested, and the policy specification to be used.
- **How:** Ignore the lifelines in the system specification that are not interacting with the chosen lifeline, and ignore the part of the system specification that does not perform any action of relevance for the policy specification.

- **Input:** A system specification, a policy specification, and the lifeline to be tested.
- **Output:** A redefined system specification, and a policy specification.
- **Tool support:** Defining a specification can be performed with any tool that has the function to draw a sequence diagram. When using the Køller-tool, which is a plugin to Eclipse [11], there are several options. We have used Papyrus [7], which is also a plugin to Eclipse.

Description The system specification is the one to be tested if it adheres to the policy specification. Since the Køller-method generates test diagrams, as sequence diagrams, and test the system specification against those, the system specification and the policy specification should be defined as sequence diagrams.

In order to test if the system specification adheres to the policy specification, we need to know which lifeline that should be tested. The lifeline has to be present in both specifications.

Given the chosen lifeline to be tested, we find what part of the system specification that we need to test. If not using the Køller-tool in the next steps, the specification must have only the chosen lifeline in the sequence diagram, this in order to, in a later step, be able to tell the Escalator what lifeline that should be tested. If using the Køller-tool in the next steps, there is no need to remove the other lifelines.

This version of the Køller-method supports the operators seq, trigger, obligation, and prohibition for the policy specification, and the operator seq in the system specification.

Tool Support This step can be conducted in any tool with the function to draw sequence diagrams, and that saves the sequence diagrams as .uml-files.

If we want to draw a sequence diagram in Eclipse [11], we need to install another plugin. We have used Papyrus [7], an open source tool that easily can be installed, but there are other options as well.

If the sequence diagrams are made using a UML-tool outside of Eclipse, the sequence diagrams have to be imported to the workspace that is going to be used with the Køller-tool, before going to the next step.

The Køller-tool will make a new system diagram from the one it gets as output from this step, which will only include the lifelines that interact with the chosen lifeline, but not remove the parts of the system specification that does not have relations to the policy specification. So the important thing here, if using the Køller-tool in the remaining steps, is to define the part of the system specification that needs to be tested.

The Køller-tool have some assumptions regarding the specifications. We have listed them in Section 7.2.2.

7.1.2 Step 2: Generate test diagrams

- **Objective:** Go through the specifications and analyze its contents, and from the results of the analysis, generate test diagrams.

- **How:** Make the alphabets of the system specification and the policy specification, and ,based on these, make the alphabets of the test diagrams. When having the alphabets of the test diagrams, find the traces it can have and draw or write the test diagrams.

  This is performed by conducting the sub-steps illustrated in Figure 7.2 and listed below.

  2.1 Make the alphabets of the system specification and the policy specification

  2.2 Make the alphabet of the test diagrams

  2.3 Make the test diagrams

- **Input:** A redefined system specification, a policy specification, and the type of the policy.

- **Output:** Test diagrams, or a result.

- **Tool support:** The Køller-tool supports this step by performing seven sub-steps as illustrated in Figure 7.3. In order for the Køller-tool to conduct the step it requires some additional information, which the Køller-method does not require.

  The additional information is the following.

  – The maximum-amount of test diagrams to be generated

    There are no limits in how big the specifications can be, so the test diagrams can exceed the expected, and wanted, size. There can also be generated more test diagrams then what might be expected and wanted. Because of this we have to specify the maximum amount of test diagrams to be generated.

    The higher this number is, the more accurate can the testing be, but also more time consuming.

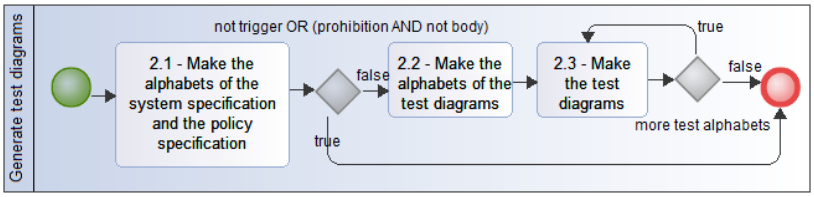


Figure 7.2: Step 2 of the Køller-method

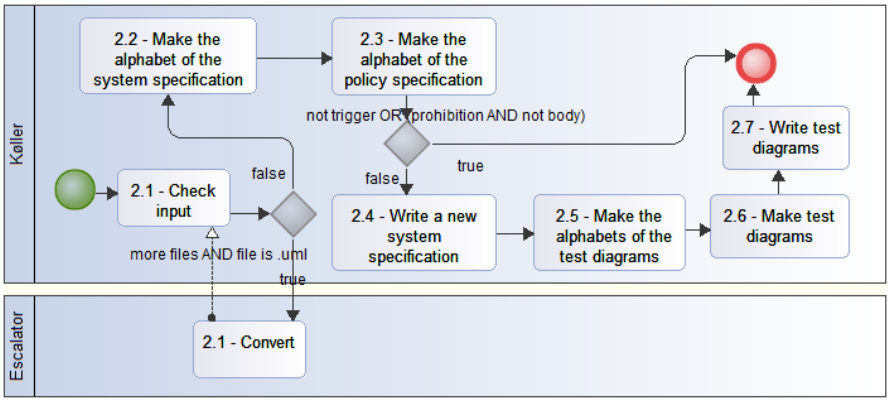


Figure 7.3: Step 2 of the Køller-tool

7.1.2.1 2.1: Make the alphabets of the system specification and the policy specification

- **Objective:** Make the alphabet of the system specification, and the alphabet of the policy specification.
- **How:** Go through the specifications, and find the messages that goes to and from the lifeline chosen for the testing.
- **Input:** A redefined system specification, a policy specification, and the type of the policy.
- **Output:** The alphabet of the system specification and the alphabet of the policy specification, or a result.
- **Tool support:** The Køller-tool supports this sub-step by, if needed, first converting the files containing the specifications, and then going through the specifications and finding the messages that goes to and from the chosen lifeline.

Description In this sub-step we find the alphabets of the specifications. The alphabet is all the messages in a sequence diagram that are sent or received at the chosen lifeline. We keep track of the name, receiver, and sender of all the messages. The ones that go to the chosen lifeline are part of the input-alphabet, and the ones from the chosen lifeline are part of the output-alphabet. This is conducted first with the system specification and then with the policy specification. We also keep track of which messages that belongs to the trigger, and which belongs to the body.

When making the alphabet of the policy specification, we also check if the messages from the trigger are present in the system specification's alphabet. We do this by checking if there is a message with the same name in the system specification's alphabet, and then checking if the message has the same receiver and sender. If there are messages in the trigger that are not in the system specification's alphabet, the trigger is not present, and there is no need for testing. The system specification then adheres to the policy specification, and this step is done. We then go to Step 4.

If all messages in the trigger are present in the system specification's alphabet, we start analyzing the body. If the policy specification has the type prohibition, we also check if the messages in the body is present in the system specification's alphabet. If there are messages in the body that are not in the system specification's alphabet, there is no need for testing. Since the policy specification is a prohibition, the system specification adheres to the policy specification. We then go to Step 4.

Tool Support This sub-step is supported by the Køller-tool with the use of three steps, which we have described below.

2.1 Convert a specification-file

The Køller-tool starts by checking the files containing the specifications. If a file is a .uml-file (in XML-format), it first converts it into a .sd-file (textual syntax of STAIRS). It does this in order to have a less complex syntax to analyze. The task is conducted by using the Escalator, which has a method for converting from XML to STAIRS textual syntax. This is done to all the .uml-files.

2.2 Make the alphabet of the system specification

The Køller-tool then analyzes the system specification. It saves all the messages that have the chosen lifeline either as its sender or receiver.

2.3 Make the alphabet of the policy specification

It then starts analyzing the trigger specification. For each message in the trigger specification, it checks if the message is present in the system specification's alphabet. If a message in the trigger is not present in the system specification's alphabet, the testing is done because the system specification then adheres to the policy specification. The Køller-tool then saves the result, and goes to Step 4.

If the messages in the trigger are present, they, with their related sender and receiver, are saved as part of the trigger alphabet.

The Køller-tool then starts analyzing the body specification. If the policy specification has the type prohibition, each message in the body is checked if it is present in the system specification's alphabet. If not all are present, the system specification adheres to the policy specification. The Køller-tool saves the result, and goes to Step 4. If all the messages are present, they are saved as part of the body-alphabet.

7.1.2.2 2.2: Make the alphabets of the test diagrams

- **Objective:** Make the alphabets of the test diagrams.
- **How:** Use the alphabets made in the previous sub-step to make the test diagram's alphabets.
- **Input:** The alphabet of the system specification and the alphabet of the policy specification.
- **Output:** The alphabets of the test diagrams.
- **Tool support:** The Køller-tool supports this sub-step by using the alphabets made in the previous sub-step to make the alphabets of the test diagrams.

Description Based on the alphabet of the system specification and policy specification, we make the alphabets of the test diagrams. The test diagram's output-alphabet will be the union [24] of the policy specification's output-alphabet and the system specification's output-alphabet. The input-alphabet will differ for the test diagrams, as mentioned in the beginning of this chapter. The first one will have the same input-alphabet as the policy specification's input-alphabet. The next test diagram will have the same input-alphabet as the previous one, plus one message from the system specification's input-alphabet. New alphabets will be successively made in this accumulative way until there are no further messages to extend with.

Tool Support This sub-step is supported by the Køller-tool, which conducts the task as defined in the Køller-method. The Køller-tool also makes a new system specification, as mentioned in Section 7.1.1, which is made before making the test diagram's alphabet.

The Køller-tool does this sub-step with the use of the two steps described below.

2.4 Write a new system specification

After the alphabet of the policy specification and the alphabet of the system specification are made, the Køller-tool makes a new system diagram. The new system diagram will be equal to the original one, with some exceptions. All lifelines that do not interact with the chosen lifeline are not included. All the lifelines that interact with the chosen lifeline are made as gates. A gate is made when a message is sent or received at the sequence diagram's frame. So we can think of the other lifelines as being *outside* of the sequence diagram. The Køller-tool represents them as gates in the new system specification in order to tell the Escalator what lifeline that should be tested.

2.5 Make the alphabets of the test diagrams

The Køller-tool will make the alphabets of the test diagrams the same way as described for the Køller-method. The Køller-tool will make as many alphabets as the user has specified, or, if it can not make that many, it will make as many as possible.

7.1.2.3 2.3: Make the test diagrams

- **Objective:** Make the test diagrams to be tested together with the system specification.
- **How:** By using the alphabets of the test diagrams made in the previous sub-step, we make the traces the test diagrams should contain, and draw the test diagrams.
- **Input:** The alphabets of the test diagrams.
- **Output:** Test diagrams.
- **Tool support:** The Køller-tool supports this sub-step by using the alphabets of the test diagrams, and make test diagrams with the desired traces.

Description In order to make the test diagrams, we need to find all the different ways the messages in the test diagram's alphabets can be combined. In other words we need to find all the orders the messages in each of the alphabets can be arranged in. Thereafter

we remove the orders where the policy-messages are not in the same order as in the policy specification. The trigger-messages have to be ordered before the body-messages, and all in the same order as in the policy specification.

All the different orders the messages can come in are all the traces that has to be in the test diagram. We then write or draw a sequence diagram that contains all those traces using STAIRS textual or graphical syntax. We make all the traces as alternatives using several alt-operators.

This sub-step is conducted with all the alphabets of the test diagrams.

Tool Support The Køller-tool supports this sub-step by further dividing it into two sub-steps, which are described below.

2.6 Make test diagrams

The Køller-tool finds all the orders the messages can be arranged in, and then removes the orders where the policy messages are not in the same order as in the policy specification.

2.7 Write test diagrams

The Køller-tool writes a sequence diagram that contains all the traces found for each test diagram in the previous step, using the STAIRS textual syntax. It makes all traces an alternative, with several alt-operators.

7.1.3 Step 3: Test the system specification against the test diagrams

- **Objective:** Test the system specification against the test diagrams in order to find out if the system specification adheres to the policy specification.
- **How:** By the use of the Escalator we generate new test diagrams from the system specification, and test them against the test diagrams we have generated. We have illustrated this step in Figure 7.4 on page 66.

  This step is performed with the use of two sub-steps listed below.

  3.1 Generate test diagrams with the use of the Escalator

  3.2 Test test diagrams from the Escalator against test diagrams we have generated
- **Input:** Test diagrams.
- **Output:** Test verdicts from the Escalator.

- **Tool support:** This step can be conducted in the Escalator, and the Køller-tool also supports this step by using the Escalator.

  The Køller-tool, however, needs some additional information in order to conduct this step:

  – The maximum-number of times a test diagram should be tested

    Each test can be run up to this number of times. The higher this number is, the more accurate is the test, but also the more time consuming.

    This number is equal to the number of test diagrams that will be generated from the system specification. We go more into details about this in Sub-step 3.1.

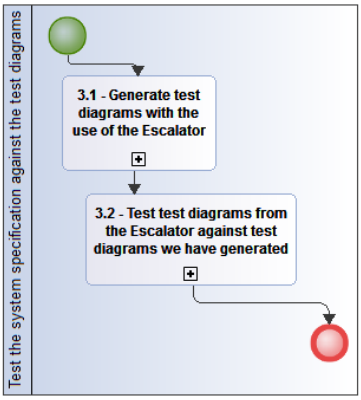


Figure 7.4: Step 3 of the Køller-method

7.1.3.1 3.1: Generate test diagrams with the use of the Escalator

- **Objective:** Generate test diagrams based on the system specification in order to be able to test with the Escalator.
- **How:** By using the Escalator to generate the test diagrams.
- **Input:** The new system specification made in Sub-step 2.4.
- **Output:** Test diagrams based on the system specification.
- **Tool support:** Both the Escalator and the Køller-tool supports this sub-step, where the Køller-tool uses the Escalator when performing it.

Description In order to test with the Escalator, we need to generate test diagrams in the Escalator. Because of this we generate test diagrams from the system specification made in Sub-step 2.4, and can then use them to conduct testing against the test diagrams we made in Step 2.

We give the system specification to the Escalator as input, which generates new test diagrams from the system specification. The traces in the new test diagrams are given verdicts telling if they are present in the original system specification or not. This is done as explained in Section 5.4.1.1.

Tool Support This sub-step is a function in the Escalator, and is described in Section 5.4.1.1. The Køller-tool uses the Escalator to perform this sub-step, and calls the method that is described in Section 5.4.1.1.

7.1.3.2 3.2: Test test diagrams from the Escalator against test diagrams we have generated

- **Objective:** Test policy adherence with the use of the Escalator.
- **How:** By using Escalator's function for testing refinement.
- **Input:** Test diagrams based on the system specification, and test diagrams generated in Step 2.
- **Output:** Test verdicts from the Escalator.
- **Tool support:** This sub-step is also supported by both the Escalator and the Køller-tool, where the Køller-tool uses the Escalator when performing it.

Description We use the Escalator to test the test diagrams from the system specification against the test diagrams we generated. The Escalator tests if there is a trace in the system specification's test diagram that is present in the test diagrams. This is done as explained in Section 5.4.1.2.

As stated in Section 5.4.1.2, if there is a trace with the verdict pass or inconclusive, the final result is pass. If the verdict is fail, it means that there is a negative trace. The final result will be fail if the test diagram from the system specification does not contain more negative traces than the negative traces that are also in the test diagrams tested against. In all other cases the final result will be pass.

These verdicts are not answering the adherence question, so in the next step we analyze the results, and make a test report with the final answers. We therefore need to keep all the results from all the tests, and have them with us into the next step.

Tool Support The Køller-tool also uses the Escalator in this sub-step, conducting it in the same way as described above, but without any user interaction. The Køller-tool conduct this sub-step for each test diagram, and collects the answers to be further analyzed in the next step.

7.1.4 Step 4: Analyze result, and make test rapport

- **Objective:** Make a report answering the policy adherence question.
- **How:** We need to analyze the results given from either Step 2, or Step 3, and from this deduce the final results. We have illustrated this step in Figure 7.5.

  This is performed with the use of the following sub-steps:

  4.1.a Analyze the result from Step 2

  4.1.b Analyze the result from Step 3

  4.2 Write the report

  The first two sub-steps are two alternative sub-steps for analyzing results from either Step 2 or Step 3.
- **Input:** Result from Step 2, or test verdicts from Step 3.
- **Output:** Result report.
- **Tool support:** The Køller-tool supports this step by analyzing the result it gets as input, and writes a report.

7.1.4.1 4.1.a: Analyze the result from Step 2

- **Objective:** Analyze the result in order to get the answer to the policy adherence question.
- **How:** By analyzing the result we get as input, which tells what message from the policy specification that where not found in the system specification.
- **Input:** Result from Step 2.

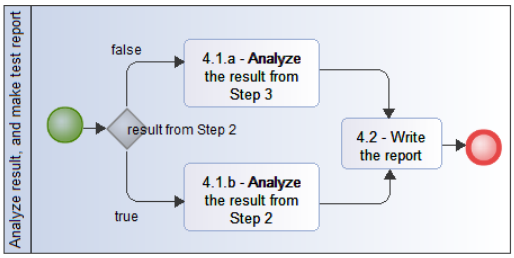


Figure 7.5: Step 4 of the Køller-method

- **Output:** Final results to the adherence question.
- **Tool support:** The Køller-tool supports this step by analyzing the input.

Description If there was a trigger-message that was not found in the alphabet of the system specification, the system specification adheres to the policy specification. This is because the policy specification is not triggered by the system specification and therefore does not apply. This result is independent of the type of the policy.

If there was a policy with the type prohibition, and a body-message was not present, the system specification is also adhering to the policy specification. This is because if not all the body-messages are present, the system specification does not have behavior that matches the prohibited behavior.

So in both cases where the result comes from Step 2, the system specification adheres to the policy specification.

Tool Support The result from Step 2 will contain one verdict, and the reason for the verdict. The Køller-tool then analyzes the result, and modifies it in a format that can be passed over to the next step, Step 4.2.

7.1.4.2 4.1.b: Analyze the result from Step 3

- **Objective:** Analyze the result in order to get the answer to the adherence question.
- **How:** By analyzing the verdicts and traces we get from the Escalator.
- **Input:** Test verdicts from the Escalator.
- **Output:** Final results to the adherence question.

Policy rule	Trigger present	Body present	Final result
obligation	yes	yes	pass
obligation	yes	no	fail
obligation	no	yes	pass
obligation	no	no	pass
prohibition	yes	yes	fail
prohibition	yes	no	pass
prohibition	no	yes	pass
prohibition	no	no	pass

Table 7.1: Overview of the desired final result, in relation to the traces from the Escalator

- **Tool support:** The Køller-tool supports this step by analyzing on the traces it gets as input.

Description The Escalator will generate some result-verdicts from the testing in Step 3.2. However, those verdicts are not answering the policy adherence question. We therefore have to read and analyze the results, and find the actual answer to our question.

The results we want are based on the traces of the verdicts, and the type of policy that is being tested. We need to take a closer look at the traces that gave the verdicts. We show the desired results, with relation to the traces, in Table 7.1.

If there is a policy with the type obligation and the traces do not contain the trigger, the system specification adheres to the policy specification since the policy does not apply. It also adheres to the policy specification if there are traces containing both the trigger and body. In all other cases the system specification does not adhere to the policy specification.

If there is a policy with the type prohibition and the traces do not contain the trigger, the system specification adheres to the policy specification, just like with a policy with the type obligation. If there are traces containing both the trigger and the body, the system specification does not adhere to the policy specification. In all other cases the system specification adheres to the policy specification.

Tool Support The result from Step 3 will contain one or more verdicts, followed by the trace that gave the verdict.

Since the Escalator's verdicts are not answering the adherence question, the Køller-tool needs to analyze the result as described above. It checks each trace, with relation to the type of policy. It then corrects the verdicts from the Escalator. It does the checking, and correction as showed in Table 7.1. If there is a policy with the type prohibition, the result

will be fail if the trigger and body are present, and in all other cases pass. If there is a policy with the type obligation, the result will be fail if the trigger is present but not the body, and pass in all other cases.

7.1.4.3 4.2: Write the report

- **Objective:** Write a report containing the results from the policy adherence testing.
- **How:** Make a report, and write the final results to it.
- **Input:** Final results.
- **Output:** Result report.
- **Tool support:** This step is fully supported by the Køller-tool by the creation of a report-file and writing the results to the file.

Description When we have analyzed the results, we make a report that contains the final results of the policy adherence testing.

If there is one fail-verdict among the final results, no matter how many pass-verdicts, the system specification does not adhere to the policy specification. The fail-verdict states that there exists a trace in the system specification which does not follow what is defined in the policy specification.

Tool Support Here the final results that comes as input are written to a *read-only* file with the extension .result. The Køller-tool will write the report in the manners showed in Figure 7.9 on page 77.

7.2 Køller-Tool

We here go through the Køller-tool, its environment, and the components it consists of. We also have a closer look at the tasks of each component, in relation to the steps of the Køller-method. The Køller-tool also has some assumptions regarding the specifications it tests that are listed in Section 7.2.2.

7.2.1 Description

The Køller-tool is, as mentioned, made as a plugin to Eclipse [11]. It also uses another plugin, the Escalator, to conduct some of the tasks. In Figure 7.6 on page 72 we have

illustrated the environment of the Køller-tool, and the communication between the different components in the environment. We have not included the arguments that are sent with the calls.

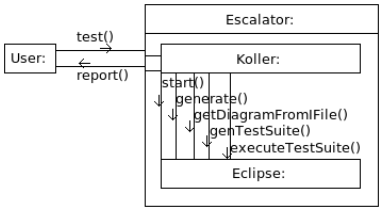


Figure 7.6: The environment of the Køller-tool

The user only interacts with Eclipse. When the user starts the Køller-tool in the menu in Eclipse, the Køller-tool requires some information, and after getting this, it starts its process. The Køller-tool then uses some methods in the Escalator to perform some of its tasks, but this is not visible to the user. The only window the user interacts with is illustrated in Figure 7.7.

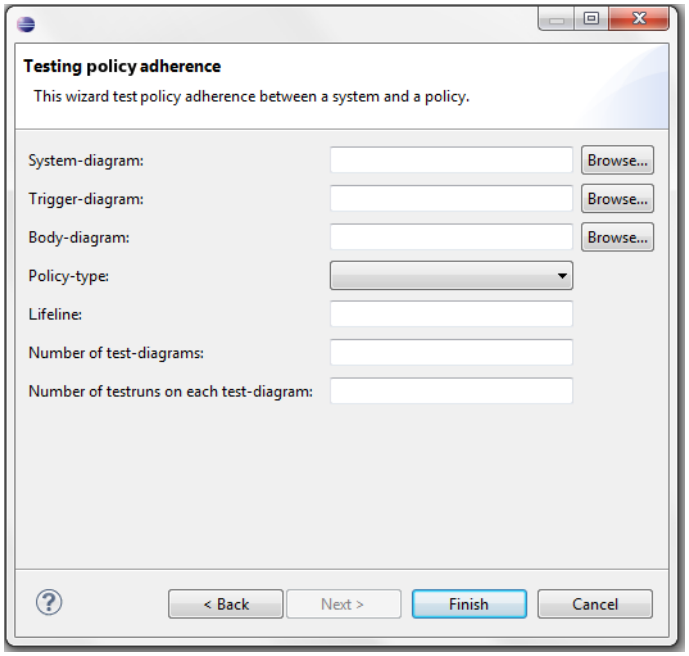


Figure 7.7: The window the user interacts with

In Figure 7.8 on page 76 we have the interaction between the user, the Køller-tool and

the Escalator in more detail. From the user's perspective, he/she only needs to start the process, and then the answer will come after some time.

At three different cases there can be found results; first in Step 2 when one or more messages from the trigger is not present, second when the policy specification has the type prohibition and one or more messages from the body is not present, and third after the testing with the use of the Escalator. We have illustrated a report from each of these cases in Figure 7.9 on page 77.

In Figure 7.10 on page 77 we have illustrated the packages, and classes the Køller-tool consist of. We have also listed the methods of the classes. There is one package for the user interface, one for the testing, and one for administrating the plugin. The package for the testing consists of different classes, each class with different tasks. Below we list the classes and state their tasks.

- Analyze

  This is where the main testing is performed. This class performs Step 2 of the Køller-tool, with the help of the Escalators converting-function in Sub-step 2.1.

- AnalyzeResult

  This class is called after the testing is done. Here, Step 4 is performed, by analyzing the result and writing a report.

- PermutationGenerator

  This class generates all the orders the messages in two alphabets can be in. The class Analyze uses this class when making the traces of the test diagrams.

- ListSeq

  This class serves as a container that is used when writing the test diagrams. The class Analyze uses this class when making the test diagrams.

In order for us to make this into a plugin we needed some additional classes. We have listed them below, with a description.

- Activator

  This class starts the plugin, and is the one that controls the plugin's life cycle.

- KollerNewWizard

This class makes the window the user interacts with when starting the Køller-tool. What is showed in the window is defined in the class KollerNewWizardPage. This class also calls the right methods when the user has pressed the finish-button.

It calls the methods in the class Analyze to perform Step 2, then if it gets a result back it calls the methods in AnalyzeResult to perform Step 4. If the Køller-tool gets test diagrams back from the methods in the class Analyze, it calls the methods in the Escalator to perform Step 3. When getting the results back, it starts the methods in the class AnalyzeResult. After the report has been written, the process is done, and the plugin is being terminated.

- KollerNewWizardPage

  In this class all the labels and boxes the user will interact with when testing with the Køller-tool is defined.

7.2.2 Assumptions Regarding Specifications

The version of the Køller-tool we have developed in this project is the first version, and is not supporting all that might be wanted from the requirements we listed in Chapter 4. We have here listed the assumptions the Køller-tool makes regarding the specifications that it gets as input, and is going to be tested.

- There can only be seq-fragments

  This version of the Køller-tool only supports the seq-operator, so all others, like opt, alt, par, etc. will be ignored.

- The policy specification has to be made with the use of two sequence diagrams, one containing the trigger, and one containing the body

  This is in order to tell the Køller-tool what specification is the trigger and what is the body.

- All messages in the same sequence diagram must have different names

  If there are messages with the same name, only one of them will be registered, and the others will be ignored.

- The messages cannot have arguments

  We have not implemented support for handling messages with arguments, so the Køller-tool will analyze the specifications wrongly if the messages have arguments.

- There has to be at least one input message either in the policy specification or in the system specification

  For the Køller-tool to be able to test policy adherence, there has to be at least one input message in one of the specifications.

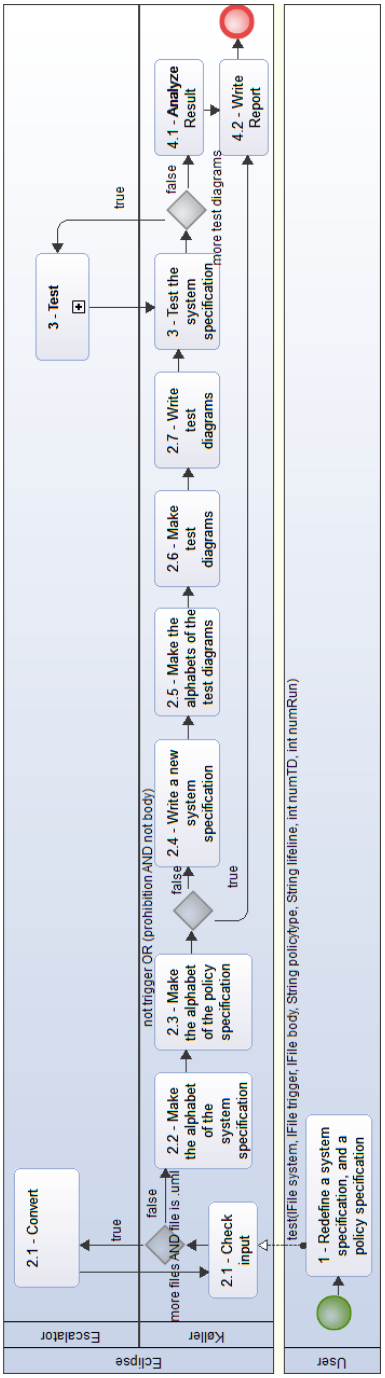


Figure 7.8: The steps of the Køller-tool

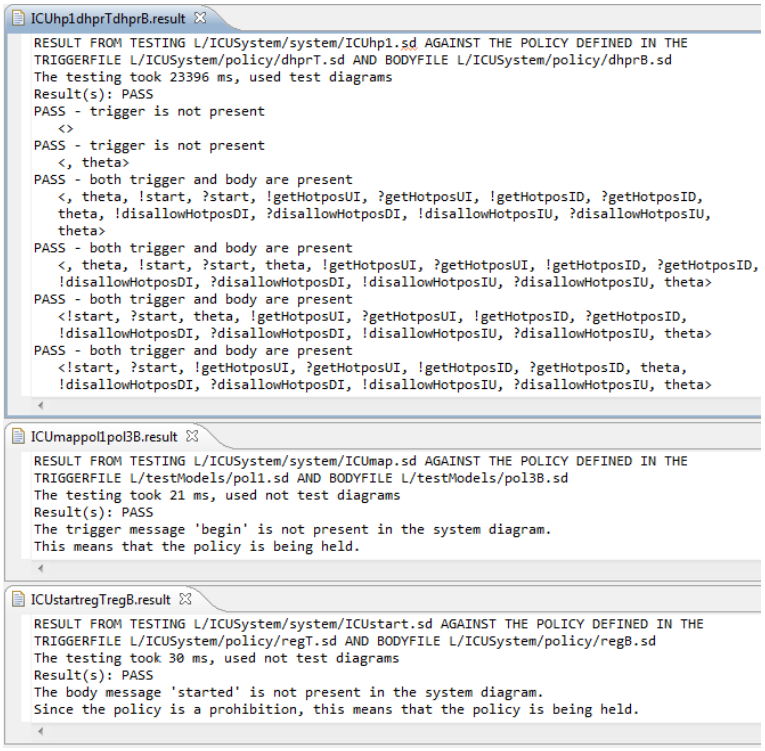


Figure 7.9: Three examples of reports from the Køller-tool

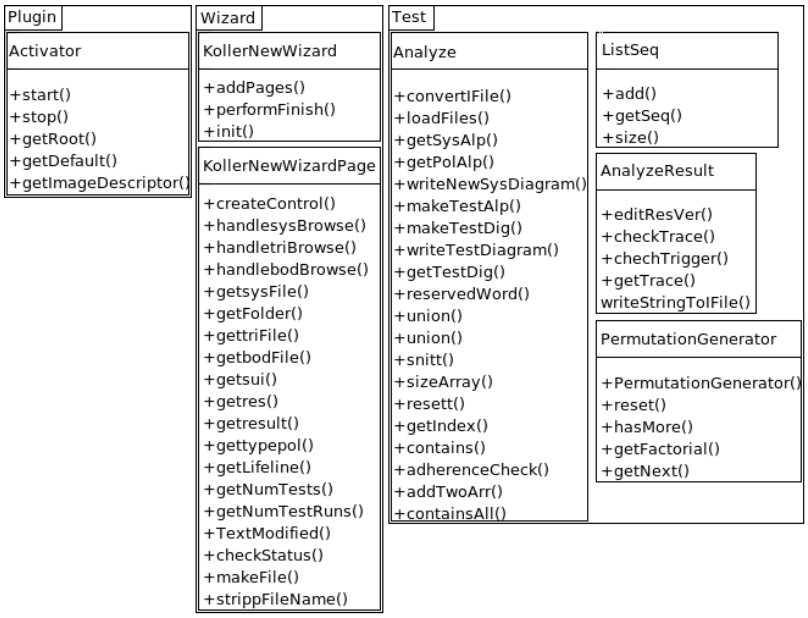


Figure 7.10: The components of the Køller-tool

Chapter 8

Evaluation of the Artifacts with Respect to the Needs

In this chapter we evaluate the developed artifacts. First we evaluate the requirements to the Køller-method by conducting the same kind of case study as we did with the Escalator in Chapter 6. The result from the evaluation of the Køller-method is found in Section 8.1. We first go through how well the Køller-method fulfills the functional requirements, and then the non-functional requirement. The requirements are found in Section 4.2.1.

We also evaluate the Køller-tool by conducting the same case study as we did with the Køller-method. The result from the evaluation is found in Section 8.2. We first go through how well the Køller-tool fulfills the functional requirements, and then the non-functional requirements. The requirements are found in Section 4.2.2.

8.1 Køller-Method

In order to evaluate the Køller-method we tested the policy specifications in Appendix C against the system specification in Appendix B. In Table 8.1 on page 80 and Table 8.2 on page 81 we show the result of the testing. We performed the steps of the Køller-method, without using the Køller-tool, in order to find out how well the Køller-method performs on its own. For each policy specifications in the tables, the tables show the system specification, lifeline tested, if they where tested with the use of test diagrams, the result, and the desired result. Table 8.1 on page 80 assume that the policy specifications have the type obligation, and Table 8.2 on page 81 that they have the type prohibition. In Appendix D we have illustrated the testing of two of the policy specifications, pol1 and pol2.

As stated in Section 7.1.4.3, if we conclude on two different verdicts, both pass and fail,

Pol	Sys	Lifeline	TD	Result	D. res
pol1	ICUstart	ICUSystem	0	pass	pass
pol1	ICUstart2	ICUSystem	0	pass	pass
pol1	ICUstart3	ICUSystem	0	pass	pass
pol2	ICUhp	ICUSystem	0	pass	pass
pol2	ICUhp1	ICUSystem	2	pass	pass
pol3	ICUstart	ICUSystem	3	fail and pass	fail
pol3	ICUstart2	ICUSystem	3	fail and pass	fail
pol3	ICUstart3	ICUSystem	3	fail and pass	fail
pol4	ICUmap	User	1	fail	fail
pol5	ICUhp	ICUSystem	0	pass	pass
pol5	ICUhp1	ICUSystem	2	fail and pass	fail
pol6	ICUconf	DB	2	fail	fail
pol6	ICUconf1	DB	2	fail	fail
pol7	ICUmap	ICUSystem	0	pass	pass
pol8	ICUmap	ICUSystem	0	pass	pass
pol9	ICUmap	User	0	pass	pass
pol10	ICUconf	DB	2	pass	pass
pol10	ICUconf1	DB	2	pass	pass

Table 8.1: The result from testing the policy specifications, with the type obligation

Pol	Sys	Lifeline	TD	Result	D. res
pol1	ICUstart	ICUSystem	0	pass	pass
pol1	ICUstart2	ICUSystem	0	pass	pass
pol1	ICUstart3	ICUSystem	0	pass	pass
pol2	ICUhp	ICUSystem	0	pass	pass
pol2	ICUhp1	ICUSystem	2	fail and pass	fail
pol3	ICUstart	ICUSystem	3	pass	pass
pol3	ICUstart2	ICUSystem	3	pass	pass
pol3	ICUstart3	ICUSystem	3	pass	pass
pol4	ICUmap	User	1	pass	pass
pol5	ICUhp	ICUSystem	0	pass	pass
pol5	ICUhp1	ICUSystem	2	pass	pass
pol6	ICUconf	DB	2	pass	pass
pol6	ICUconf1	DB	2	pass	pass
pol7	ICUmap	ICUSystem	0	pass	pass
pol8	ICUmap	ICUSystem	0	pass	pass
pol9	ICUmap	User	0	pass	pass
pol10	ICUconf	DB	2	pass	pass
pol10	ICUconf1	DB	2	pass	pass

Table 8.2: The result from testing the policy specifications, with the type prohibition

it means that there is one or more traces in the system specification that is adhering to the policy specification, and that there is one or more that is not adhering to the policy specification. The final result will then be fail.

If we have a look at the tables, all tests got the desired result. With these tests we made all possible test diagrams, and therefore also got a more accurate answer then if we had made fewer. There are also a lot of the tests where there where no need to make test diagrams. The reason for this is described in Section 7.1.2.

Functional Requirements

Here we list the functional requirements from Section 4.2.1, and discuss how well the Køller-method fulfill them.

Requirement 1 *support testing policy adherence between a system specification and a policy specification*

All the tests we conducted gave the desired result, as stated in Table 8.1 on page 80 and Table 8.2 on page 81. The tests we conducted where relatively small, not that many, and we made most of them ourselves. With respect to small specifications we have strong indications that the Køller-method fulfills this requirement. In order to know if it fulfills this requirement in all cases, it has to undergo a deeper evaluation.

When defining the specifications to test, there are some restrictions on the operators that can be used. For a method to be really useful in practice, it needs to support most of the operators in the UML-standard. The most important operators will be alt, loop, neg, assert, and par.

Requirement 2 *facilitate the policy adherence test by generating test diagrams based on a system specification and a policy specification*

From the evaluation we conducted we have that 55,5 percent of the tests done with policy specifications with the type obligation, will generate test diagrams. In the case of tests done with policy specifications with the type prohibition, we have that 27,7 percent will generate test diagrams.

For all cases where the trigger where not present, the Køller-method did not generate test diagrams. The cases where the policy specification had the type prohibition, and a body message where not present in the system specification there where also not generated any test diagrams. From this we have good indications that the Køller-method fulfills this requirement.

Non-Functional Requirements

Here we discuss how the Køller-method fulfill its non-functional requirements from Section 4.2.1.

Requirement 3 *be a well defined method*

Each step of the Køller-method is defined with its input, output, and a description. We have also two examples in Appendix D, which shows the steps. We assume our stakeholders are people that have experience with modeling, and testing, as stated in Section 4.2.1. We believe the Køller-method described in Section 7.1 is well defined with regards to this kind of stakeholders. However we would need a deeper evaluation in order to know for sure that the Køller-method is well enough defined for being used daily.

8.2 Køller-Tool

We tested the policy specifications in Appendix C against the system in Appendix B, like for the Køller-method. Appendix B describe the ICU-system also with diagrams fulfilling the assumptions described in Section 7.1.1 that the Køller-tool has on specifications to be tested. In Appendix E we have examples of the testing of two policy specifications, pol1 and pol2, against the ICU-system.

In Table 8.3 on page 84 we show the result after testing the policy specifications with the type obligation. The table contains information about the policy specification and system specification tested, what lifeline where tested, the desired result, the results written to the .result-file, whether they where tested with the use of test diagrams, and the time used given in milliseconds.

We also tested all the policy specifications with the type prohibition, and got the result showed in Table 8.4 on page 85.

We chose to generate maximum five test diagrams in each test, and have five test runs. From the tables we see there where no tests that made that many test diagrams, so all tests therefore did make as many test diagrams as possible. If we had chosen to generate only one test diagram with some of the policy specifications, we would have ended up with the wrong answer in the .result-file. Below we have listed both the system and policy specification, and the type of the policy specification that gave the wrong answer, with only generating one test diagram.

- pol2, with the type obligation, tested against ICUhp1
- pol3, with the type obligation, tested against start

Pol	Sys	Lifeline	D. res	Result	TD	Time (ms)
pol1	ICUstart	ICUSystem	pass	pass	0	22
pol1	ICUstart2	ICUSystem	pass	pass	0	30
pol1	ICUstart3	ICUSystem	pass	pass	0	27
pol2	ICUhp	ICUSystem	pass	pass	0	90
pol2	ICUhp1	ICUSystem	pass	pass	2	23396
pol3	ICUstart	ICUSystem	fail	fail and pass	3	32953
pol3	ICUstart2	ICUSystem	fail	fail and pass	3	32803
pol3	ICUstart3	ICUSystem	fail	fail and pass	3	32714
pol4	ICUmap	User	fail	fail	1	541
pol5	ICUhp	ICUSystem	pass	pass	0	30
pol5	ICUhp1	ICUSystem	fail	fail and pass	2	24130
pol6	ICUconf	DB	fail	fail	2	29479
pol6	ICUconf1	DB	fail	fail	2	2282
pol7	ICUmap	ICUSystem	pass	pass	0	21
pol8	ICUmap	ICUSystem	pass	pass	0	42
pol9	ICUmap	User	pass	pass	0	22
pol10	ICUconf	DB	pass	pass	2	25432
pol10	ICUconf1	DB	pass	pass	2	2032

Table 8.3: The result from testing the policy specifications, with the type obligation

Pol	Sys	Lifeline	D. res	Result	TD	Time (ms)
pol1	ICUstart	ICUSystem	pass	pass	0	50
pol1	ICUstart2	ICUSystem	pass	pass	0	30
pol1	ICUstart3	ICUSystem	pass	pass	0	70
pol2	ICUhp	ICUSystem	pass	pass	0	60
pol2	ICUhp1	ICUSystem	fail	pass and fail	2	25324
pol3	ICUstart	ICUSystem	pass	pass	0	30
pol3	ICUstart2	ICUSystem	pass	pass	0	40
pol3	ICUstart3	ICUSystem	pass	pass	0	20
pol4	ICUmap	User	pass	pass	0	30
pol5	ICUhp	ICUSystem	pass	pass	0	20
pol5	ICUhp1	ICUSystem	pass	pass	0	30
pol6	ICUconf	DB	pass	pass	2	33470
pol6	ICUconf1	DB	pass	pass	2	2230
pol7	ICUmap	ICUSystem	pass	pass	0	20
pol8	ICUmap	ICUSystem	pass	pass	0	20
pol9	ICUmap	User	pass	pass	0	10
pol10	ICUconf	DB	pass	pass	2	27310
pol10	ICUconf1	DB	pass	pass	2	1900

Table 8.4: The result from testing the policy specifications, with the type prohibition

- pol3, with the type obligation, tested against start3
- pol5, with the type obligation, tested against ICUhp1
- pol3, with the type prohibition, tested against ICUhp1

They all gave the wrong answer because the test diagrams did not include enough messages, a problem also described in the beginning of Chapter 7.

Functional Requirements

We here discuss how well the Køller-tool fulfill its functional requirement.

Requirement 4 *support the method for testing policy adherence*

When we compare Table 8.1 and Table 8.2 from page 80 and page 81, from the evaluation of the Køller-method without using the Køller-tool, with Table 8.3 and Table 8.4 from page 84 and page 85, from the evaluation of the Køller-tool, we see they are equal. The Køller-tool supports the steps in the Køller-method, with respect to the making of test diagrams, and analyzing the result. From Chapter 7, where we present the Køller-method and Køller-tool we see that the steps of the Køller-method are implemented in the Køller-tool.

When using the Køller-tool the policy specification has to be made with the use of two sequence diagrams, but with the use of the Køller-method it can be defined with only one. There should be possible to draw a policy diagram with Deontic STAIRS, and with the use of one sequence diagram, in the Køller-tool.

Although we are not able to define the policy specification in the same manner, we nevertheless have good indications that the Køller-tool supports the steps of the Køller-method, with some exception at Step 1.

Non-Functional Requirements

We list the non-functional requirements here, and discuss how well the Køller-tool fulfill them.

Requirement 5 *be user-friendly*

As stated we assume our stakeholder have experience with modeling and testing. We also assume they have experience with the tool Eclipse [11], or a similar one.

Our tool is a plugin to Eclipse, and the menu is a wizard, which has a standard, made by Eclipse. The wizard looks like illustrated in Figure 7.7 on page 72, and the user does not see the Køller-tool work, just get a result some time after starting the test.

Since we assume the stakeholders have experience with Eclipse, and we have made the Køller-tool using Eclipse-standard, we argue this requirement is fulfilled.

Requirement 6 *be efficient*

From Table 8.3 on page 84 and Table 8.4 on page 85 we see there are some difference in the time used for each test. When generating test diagrams, the Køller-tool uses more time.

We did not focus on this requirement, but we have done some choices regarding the design to make it efficient. If the Køller-tool already in Step 2, before generating test diagrams, has the answer of the policy adherence, it then writes the report, and terminates. We also tried several data structures to save different information while analyzing the specifications, generating the test diagrams, and concluded with the data structures that were most efficient.

However, the times used for testing the small specifications in our evaluation are too big for a tool like this. In a future version the time used when testing has to decrease.

There are several improvements we can do to the Køller-tool. One is with the task of generating the traces to the test diagrams, described in section 7.1.2.3. First all possible traces are made, for then to remove some. If we only made the ones we wanted, we would save time.

The biggest improvement when it comes to efficiency however, is Step 3, described in Section 7.1.3. Since the Escalator is not made for testing policy adherence, we had to do some adaptions in our method, and tool, for it to be able to use the Escalator. There are also some calculations conducted in this step there is no need for.

We generate test diagrams from the system specification, and there is no need for it in order to test policy adherence. We do it in order to use the testing functionality in the Escalator. The Escalator's test functionality requires test diagrams on a format we did not manage to make, in order to test against another sequence diagram.

An improvement would be that we could give our test diagrams to the Escalator, and that it could convert those test diagrams in the format it requires. Then we could test the test diagrams generated in the Køller-method together with the system specification, and get a result where we might not have to check the traces. Since we have not had the opportunity to test this improvement, we do not know how the result would answer the adherence question, but it would have been an improvement not to generate test diagrams where there is no need.

Another improvement regarding Step 3 is the actual testing functionality in the Escalator. A future version of our tool might take what can be used from the Escalator's testing functionality, and adapt it to test policy adherence. We might then get an answer from the Escalator that does not need further analyzing.

This requirement is not fulfilled with this version of the Køller-tool, and has to be taken into account in a next version.

8.2.1 Requirements for the Editor

In Section 4.2.2.1 we made some functional requirements for the editor of the Køller-tool. Here we have a look at how well the Køller-tool fulfills them.

Requirement 7 *support creating, modifying and deleting of sequence diagrams describing a system specification*

This can be done textually in the Køller-tool. In order to draw the sequence diagrams in Eclipse [11] we need to install an additional plugin, like Papyrus [7]. So this requirement is fulfilled with the use of an additional plugin.

Requirement 8 *support creating, modifying and deleting of sequence diagrams describing a policy specification*

If we split the policy diagram in two, like we illustrated in Appendix C, policy specifications can be defined textually in the Køller-tool. In order to draw the sequence diagrams in Eclipse [11] one needs to install an additional plugin, like Papyrus. So this requirement is fulfilled with the use of an additional plugin, and a change in the policy diagram.

Requirement 9 *support the viewing of a read-only text document*

This requirement is fulfilled since the Køller-tool makes a *read-only* document with the result that can be opened by the user. We have examples of some reports in Figure 7.9 on page 77.

When, however, one tries to edit the document in Eclipse, one gets an option to remove the *read-only* property.

8.2.2 Requirements for the Test-Generator

In Section 4.2.2.2 we made one functional requirement to the test-generator, and will here discuss how well we have fulfilled it with the Køller-tool.

Requirement 10 *conclude on an answer that answers the policy adherence question*

If there is a test that does not generate all test diagrams possible, the answer might not be correct, as we describe in the beginning of Section 8.2. However, as we see from the evaluation above, when tested all test diagrams possible, the results is answering the policy adherence question correctly.

8.2.3 Requirements for the Report-Generator

Here follows a discussion on how well the Køller-tool fulfill the non-functional requirements of the report-generator.

Requirement 11 *be logical*

The report generated will contain a summary of the results, and either both verdicts and traces, verdicts and reasoning for them, or verdicts, traces, and reasoning for the traces. The report first shows the summary, then the verdict and the traces or reasoning. This way the tester can, at first glimpse, see how the testing went.

Requirement 12 *contain all, and only, the information needed*

The report contains a summary of all the results found, it also contains verdicts with its trace and reasoning, or one of them, as illustrated in Figure 7.9 on page 77, and showed in Appendix E. The summary will be the different verdicts that were found under the analysis of the final result, and tell if the system specification adheres to the policy specification or not. The verdicts are based on the following trace or reasoning, or both. The trace shows where the Køller-tool got the verdict from, and the reasoning presents a description of the reason for the verdict. This is enough to find out if the system specification adheres to the policy specification, and if not, why not.

Chapter 9

Conclusion

In this chapter we summarize our project, and give some conclusions based on the development and evaluation done in Chapter 7 and 8. In Section 9.1 we present our main achievements, and in Section 9.2 discuss our suggestions for improvements to our developed artifacts, and suggest areas of future work.

In this project we aimed at making a tool-based method for testing policy adherence. We wanted our method to conduct the testing in a manner where we used test diagrams. The test diagrams would be generated based on two specifications; a policy specification and a system specification. The test diagrams would then be tested together with the system specification and give an answer regarding whether the system specification adheres to the policy specification. We also wanted the method to utilize the functionality provided by the Escalator [27] [28].

We presented our developed artifacts, the Køller-method and Køller-tool, in Chapter 7, where we went through the steps of the Køller-method, and how it is supported by the Køller-tool. The evaluation of those two artifacts, with respect to the requirements in Chapter 4, is presented in Chapter 8.

We found that the Køller-method and Køller-tool test policy adherence. They also facilitate the testing by generating test diagrams, which later are tested together with the system specification. However, there are assumptions on the specifications to be tested, listed in Section 7.2.2, that prevents the Køller-method and Køller-tool to test realistic system specifications.

There is also an issue with the time used when testing. However from our evaluation, that tested small specifications, the Køller-method does test policy adherence, and it does it in the manner we want it to.

9.1 Main Achievements

We managed to develop a method for testing policy adherence, and develop a tool for supporting the method. The tool supports the steps of the method, and gives an answer to the policy adherence question.

The test diagrams generated are made in a manner that makes them suitable for testing policy adherence, and saves time. For a system specification that has alt-operators, our tool will be better in performance than it was in the examples presented in this project. This since with alt-operators there is more chance for a smaller test diagram to find the correct answer on the policy adherence question.

9.2 Future Work

As stated in Chapter 8, there are a lot of improvements that can be done to our method, and tool. Below we discuss the improvements we mean are the most important ones, and suggestions for future work.

In order for the Køller-tool to be used it needs to be improved so the assumptions listed in Section 7.2.2 are implemented. The Køller-tool has to be able to analyze specifications that are made using standard UML. When the Køller-tool has been improved with this, it is a more suitable tool for testing policy adherence.

The next improvement has to do with the Køller-tool being efficient, which we mentioned in Section 8.2. We have used the Escalator for testing, but there might be better solutions. The Escalator is made for testing refinement, and is not adapted for testing policy adherence.

As stated in Section 2.3.2 we have not included the policy rule permission. This is also an area that can be addressed in a future version of the Køller-tool.

These are the areas which should get the most attention if a new version of the Køller-tool is to be developed.

Bibliography

[1] Altova. UML sequence diagrams. Website, accessed 20.02.2012. http://www.altova.com/umodel/sequence-diagrams.html.

[2] Arctern. Development methodologies. Website, accessed 30.04.2012. http://www.arctern.com/Development_Methodologies.aspx.

[3] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing. Using the UML Testing Profile*. Springer, 2008.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide - Second edition*. Addison-Wesley, 2005.

[5] Raouf Boutaba and Issam Aib. Policy-based management: A historical perspective. *Proceedings in Network and Systems Management*, 15(4):447–480, 2007.

[6] Anders Brunland, Knut Hegna, Ole Christian Lingjærde, and Arne Maus. *Rett på Java*. Universitetsforlaget, 2005.

[7] CEA. Open source tool for graphical UML2 modeling. Website, accessed 08.08.2011, 2006. http://www.papyrusuml.org/.

[8] Microsoft Corporation. Word 2010. Website, accessed 21.03.2012. http://office.microsoft.com/nb-no/word/?CTT=97.

[9] Hans-Erik Erikkson and Magnus Penker. *UML Toolkit*. John Wiley & Sons inc, 1998.

[10] Pavel Hruby et al. Sequence diagrams. Website, accessed 17.04.2012. http://www.uml-diagrams.org/sequence-diagrams.html#combined-fragment.

[11] The Eclipse Foundation. Eclipse. Website, accessed 10.01.2012. http://www.eclipse.org/.

[12] Falk Fraikin and Thomas Leonhardt. SeDiTeC - testing based on sequence diagrams. *Proceedings of the International Conference on Automated Software Engineering*, pages 261–266, 2002.

[13] Christopher Z. Garrett. Software modeling introduction: What do you need from a modeling tool? 2003.

[14] Gliffy. Create great looking diagrams now. Website, accessed 20.02.2012. http://www.gliffy.com/.

[15] Gunnar Gurholt and Thor E. Hasle. *Grunnleggende Systemutvikling*. Cappelen Akademisk Forlag, 2003.

[16] Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson, and Peter Williams. *Software Testing - An ISB Foundation*. BCS, 2007.

[17] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Proceedings of Software and Systems Modeling*, 4(4):355–357, 2005.

[18] Øystein Haugen and Jon Oldervik. From sequence diagrams to java-STAIRS aspects. *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pages 99–110, 2009.

[19] Manuel Hilty, Alexander Pretschner, David Basin, and Christian Schaefer. A policy language for distributed usage control. *Proceedings of the 12th European Symposium on Research in Computer Security*, pages 531–546, 2007.

[20] Pavel Hruby. Sequence diagram. Website, accessed 08.02.2012, 2010. http://www.uml-diagrams.org/sequence-diagrams-reference.html.

[21] Universitetet i Oslo. INF5150 - unassailable IT-systems. Website, accessed 01.03.2012. http://www.uio.no/studier/emner/matnat/ifi/INF5150/index-eng.xml.

[22] IBM. Sequence comparison algorithm. Website, accessed 22.02.2012. http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp.

[23] Hanov Solutions Inc. Web sequence diagrams. Website, accessed 20.02.2012. http://www.websequencediagrams.com/.

[24] HighPoints Learning Inc. Union of sets. Website, accessed 20.04.2012. http://www.icoachmath.com/math_dictionary/Union_of_Sets.html.

[25] Yanic Inghelbrecht. The fast UML sequence diagram editor. Website, accessed 20.02.2012. http://www.tracemodeler.com/index.html.

[26] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. *Proceedings of the sixteenth National Conference on Artificial Intelligence and the eleventh Innovative Applications of Artificial Intelligence Conference*, pages 291–298, 1999.

[27] Mass Soldal Lund. *Operational Analysis of Sequence Diagram Specifications.* PhD thesis, University of Oslo, 2008.

[28] Mass Soldal Lund and Fredrik Seehusen. Escalator tool, v.1.5.5. Website, accessed 11.04.2012, 2009. http://heim.ifi.uio.no/~ massl/escalator/Escalator-Tool_v1.5.5.pdf.

[29] Ken Lunn. *Software Development with UML.* Palgrave MacMillian, 2003.

[30] Lu Luo. Software testing techniques - technology maturation and research strategy.

[31] James N. Martin. *Systems Engineering Guidebook: A Process for Developing Systems and Products.* CRC Press LLC, 1997.

[32] Editors of The American Heritage Dictionaries. *The American Heritage, Dictionary of the English Language.* Houghton Mifflin Company, fourth edition, 2000.

[33] Tiwebb Web Publishing. 20 famous software disasters. Website, accessed 09.02.2012, 2000. http://www.devtopics.com/20-famous-software-disasters/.

[34] Marc Roper. *Software Testing.* McGraw-Hill Book Company, 1994.

[35] Ragnhild Kobro Runde. *STAIRS - Understanding and Developing Specifications Expressed as UML Interaction Diagrams.* PhD thesis, University of Oslo, 2007.

[36] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. The pragmatics of STAIRS. *Proceedings in Formal Methods for Components and Objects*, 4111:88–114, 2006.

[37] Philip Samuel and Anju Teresa Joseph. Test sequence generation from UML sequence diagrams. *Proceedings of Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2008.

[38] Monalisa Sarma, Debasish Kundu, and Rajib Mall. Automatic test case generation from UML sequence diagram. *Proceedings of International Conference on Advanced Computing and Communications*, 2007.

[39] Fredrik Seehusen, Bjørnar Solhaug, and Ketil Stølen. Adherence preserving refinement of trace-set properties in STAIRS: exemplified for information flow properties and policies. *Proceedings in Journal of Software and Systems Modeling*, 8(1):45–65, 2009.

[40] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design - Beyond Human-Computer Interaction.* West Sussex:John Wiley & Sons inc, 2009.

[41] Morris Sloman and Emil Lupu. Security and management policy specification. *Proceedings in IEEE Network*, 16(2):10–19, 2002.

[42] Effexis Software. Sequence diagram editor. Website, accessed 20.02.2012. http://www.sequencediagrameditor.com/.

[43] Pacestar Software. Pacestar UML diagrammer. Website, accessed 20.02.2012. http://www.pacestar.com/uml/index.html.

[44] Dehla Sokenou and Gebit Solutions Gmbh. Generating test sequences from UML sequence diagrams and state diagrams.

[45] Bjørnar Solhaug. *Policy Specification Using Sequence Diagrams.* PhD thesis, Universitetet i Bergen, 2009.

[46] Andrew Stellman. Understanding nonfunctional requirements. Website, accessed 28.02.2012, 2010. http://broadcast.oreilly.com/2010/02/nonfunctional-requirements-how.html.

[47] Perdita Stevens. XMI and MOF: a mini-tutorial. *Proceedings of the XML Technologies and Software Engineering (XSE2001)*, 2001.

[48] Markus Strauch. Quick sequence diagram editor. Website, accessed 20.02.2012. http://sdedit.sourceforge.net/.

[49] Rachel Thompson. Stakeholder analysis. Website, accessed 21.02.2012. http://www.mindtools.com/pages/article/newPPM_07.htm.

[50] Jeffrey J. P. Tsai, Bing Li, and Alan Liu. Modeling and parallel evaluation of nonfunctional requirements using FRORL requirements language. *Proceedings in Eighteenth Annual International Computer Software and Applications Conference*, pages 11–16, 1994.

[51] Allen B. Tucker, W. James Bradley, Robert D. Cupper, and David K. Garnick. *Fundamentals of computing I.* McGraw-Hill, inc, 1992.

[52] Defense Acquisition University. *Systems Engineering Fundamentals*. Defense Acquisition University, 2001.

[53] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. *Software Testing, Verification and Reliability*, 2006.

[54] Wikipedia. Spiral model. Website, accessed 21.02.2012. http://en.wikipedia.org/wiki/Spiral_model.

[55] Dan Pilone with Neil Pitman. *UML 2.0 in a nutshell*. O'Reilly, 2005.

Appendix A

Overview of Operators Usable in UML-Sequence Diagrams

Here we have illustrated examples of all the operators usable in sequence diagrams when using standard UML. We have all figures from [20].

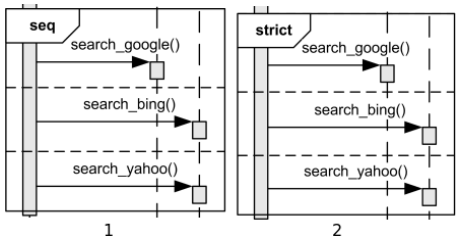


Figure A.1: 1: seq 2: strict

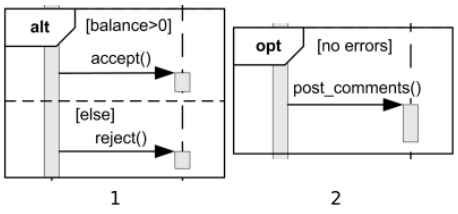


Figure A.2: 1: alt 2: opt

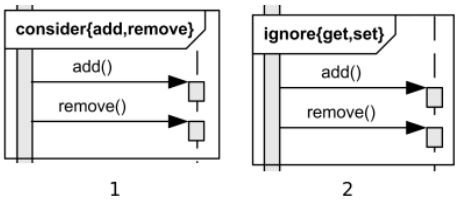


Figure A.3: 1: consider 2: ignore

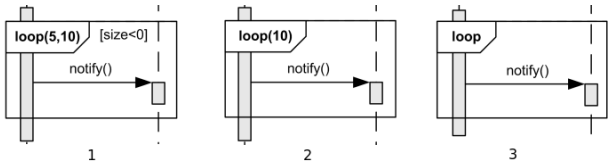


Figure A.4: 1: loop with a bound and upper guard 2: loop with a fixed guard 3: loop with no guard, an infinite loop

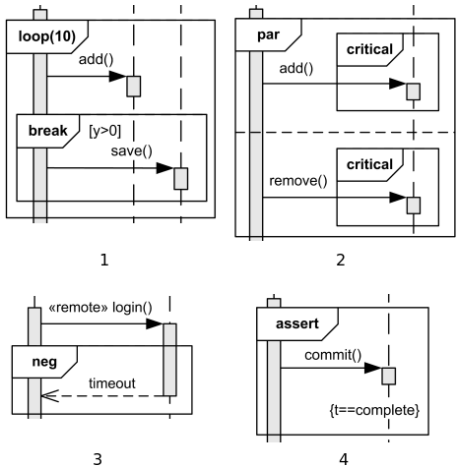


Figure A.5: 1: break 2: critical 3: neg 4: assert

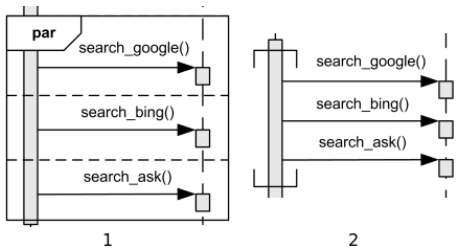


Figure A.6: 1: par 2: par coregion

Appendix B

ICU

The services the ICU-system provides:

1. register
2. getHotpos(username)
3. showMap
4. (confirmHotposRequest(username))

Figure B.1- B.13 will illustrate the ICU-system, by the use of sequence diagrams.

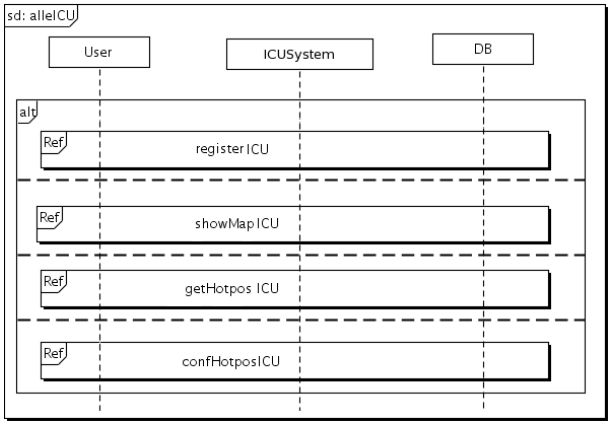


Figure B.1: The commands in ICU-perspective

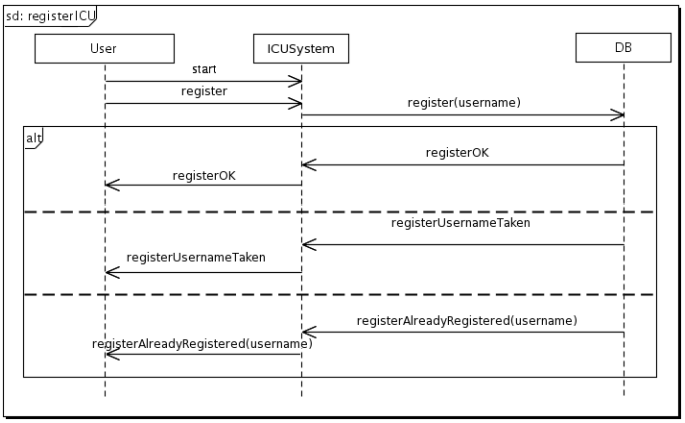


Figure B.2: The register-command in ICU-perspective

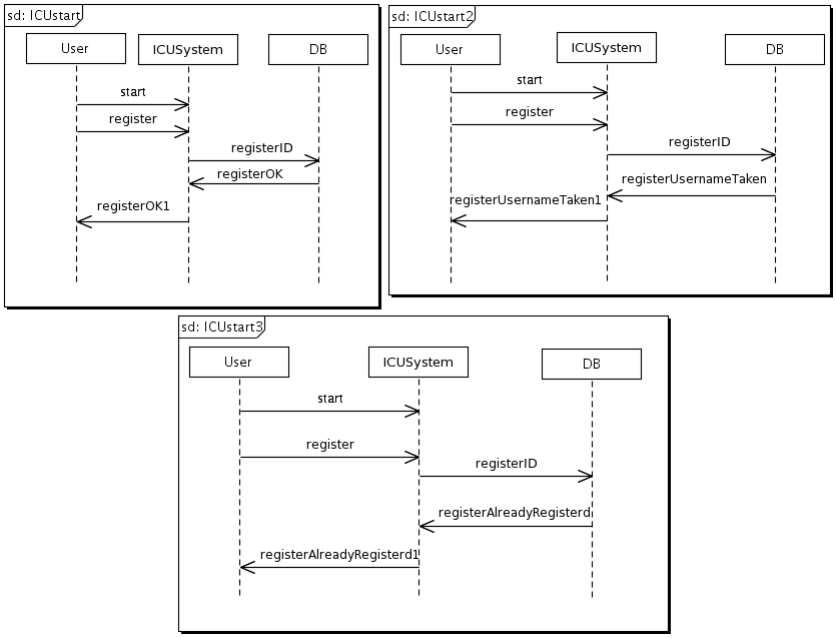


Figure B.3: The register-command in ICU-perspective, which can be given to the Køller-tool

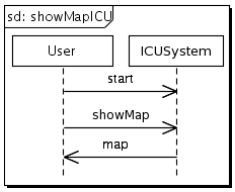


Figure B.4: The showMap-command in ICU-perspective, which can be given to the Køller-tool

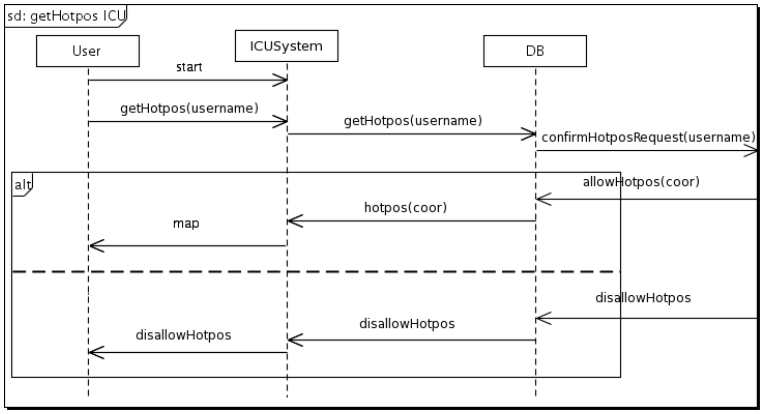


Figure B.5: The getHotpos-command in ICU-perspective

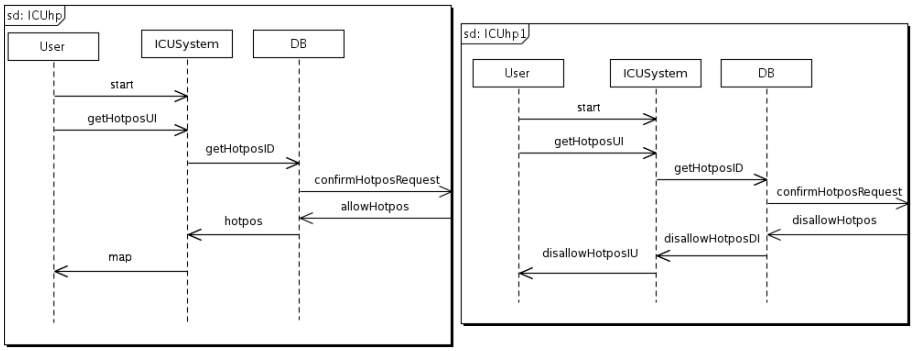


Figure B.6: The getHotpos-command in ICU-perspective, which can be given to the Køller-tool

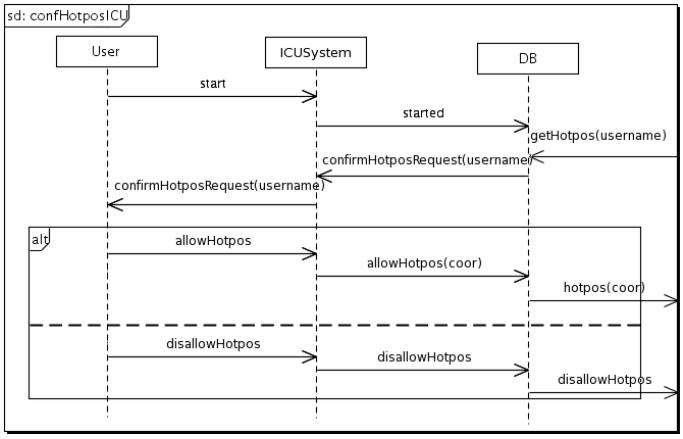


Figure B.7: The confirmHotposRequest-command in ICU-perspective

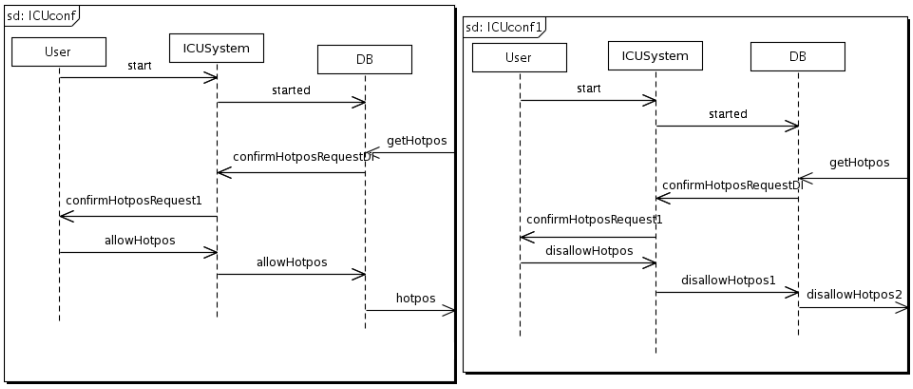


Figure B.8: The confirmHotposRequest-command in ICU-perspective, which can be given to the Køller-tool

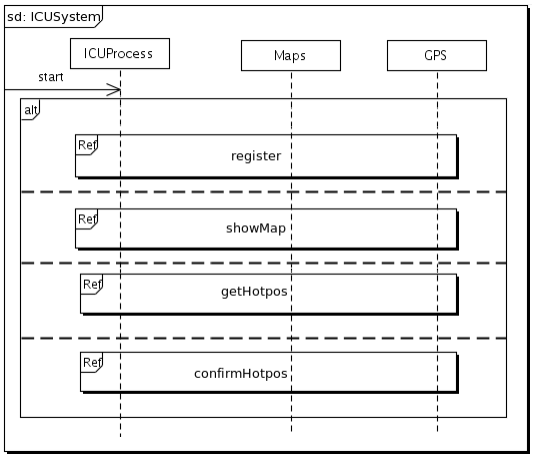


Figure B.9: The commands in ICUSystem-perspective

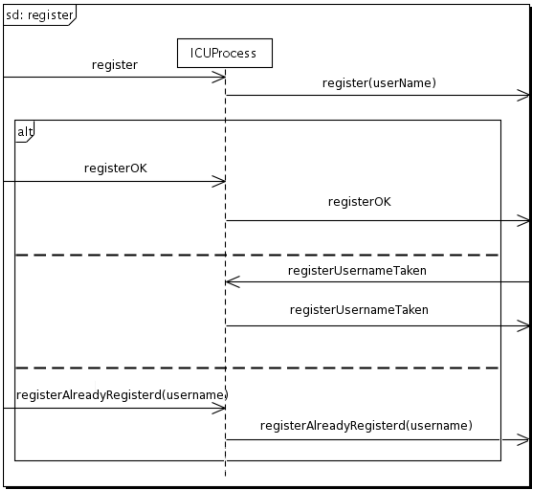


Figure B.10: The register-command in ICUSystem-perspective

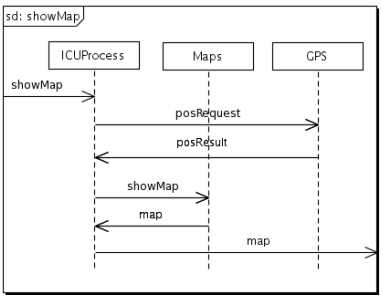


Figure B.11: The showMap-command in ICUSystem-perspective

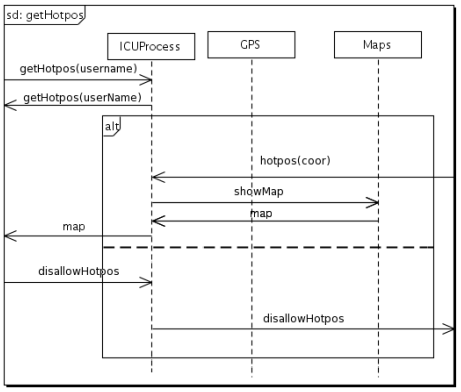


Figure B.12: The getHotpos-command in ICUSystem-perspective

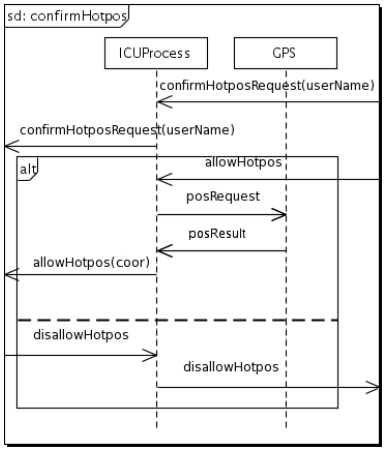


Figure B.13: The confirmHotposRequest-command in ICUSystem-perspective

Appendix C

ICU-Policies

When evaluate our developed artifacts we want to evaluate them as deep as we can. We therefore have tried to come up with all different cases there can be between a system specification and a policy specification, and made policy specifications based on those cases. We have the case where the policy should apply, and not, and that the system specification adhere to it and not. Those cases could happen for different reasons.

All the policy specifications are made for being tested on the ICU-system defined in Appendix B. We have illustrated all policy specification with the use of Deontic STAIRS and STAIRS. All the illustrations of the policy specifications have the Deontic STAIRS syntax to the left, and the STAIRS syntax to the right. With STAIRS the policy specifications are defined with the use of two sequence diagrams. The above one containing the trigger, and the below one the body. We use body instead of obligation and prohibition in the sequence diagrams with Deontic STAIRS, and that since we will use them with both types.

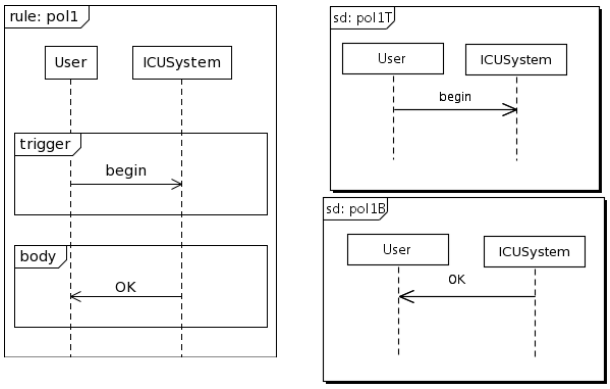


Figure C.1: Both trigger and body are not present in the system specification

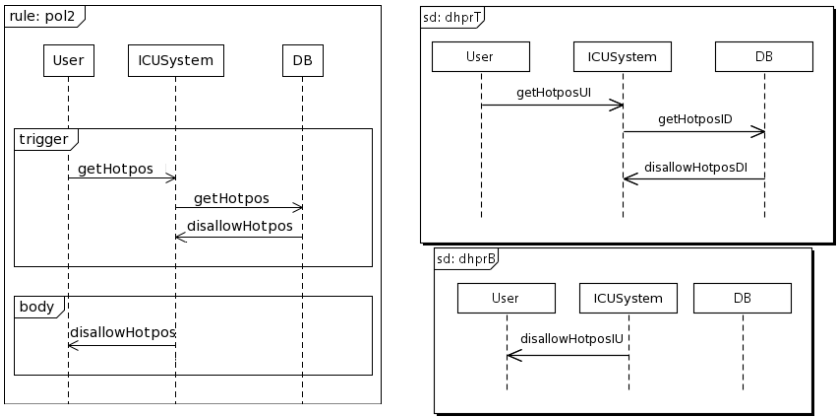


Figure C.2: Both trigger and body are present in the system specification

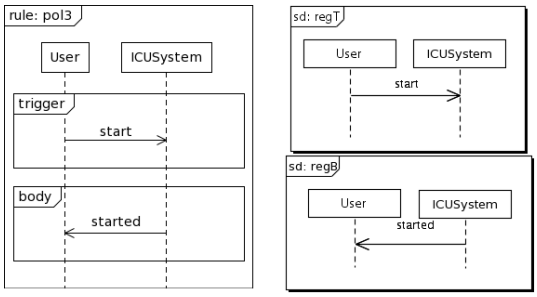


Figure C.3: Trigger is present in the system specification, but not body

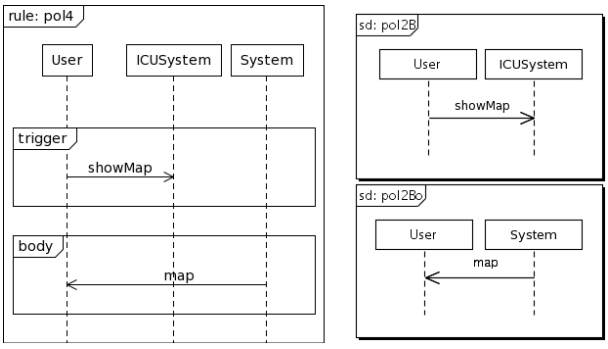


Figure C.4: Trigger is present in the system specification, but one body message has another sender than in the system specification

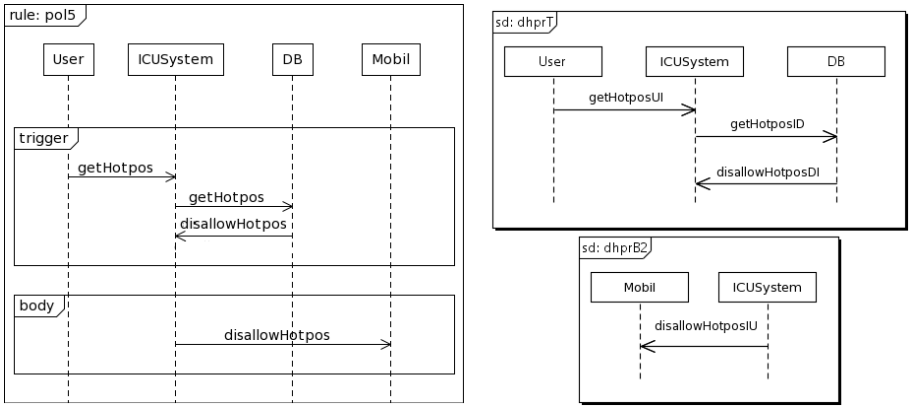


Figure C.5: Trigger is present in the system specification, but one body message has another receiver than in the system specification

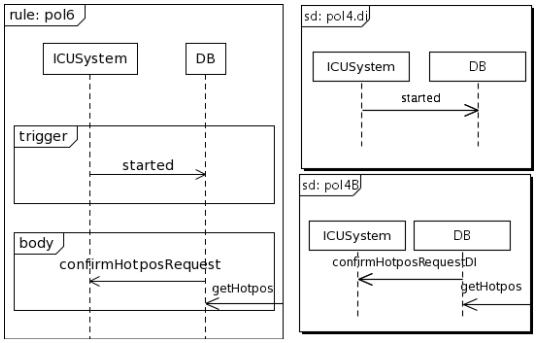


Figure C.6: Trigger is present in the system specification, but the body messages are not in the same order as in the system specification

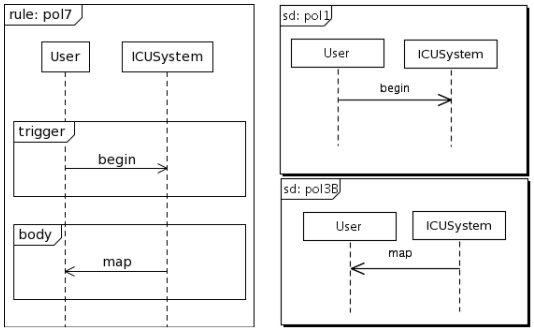


Figure C.7: Trigger is not present in the system specification, but the body is

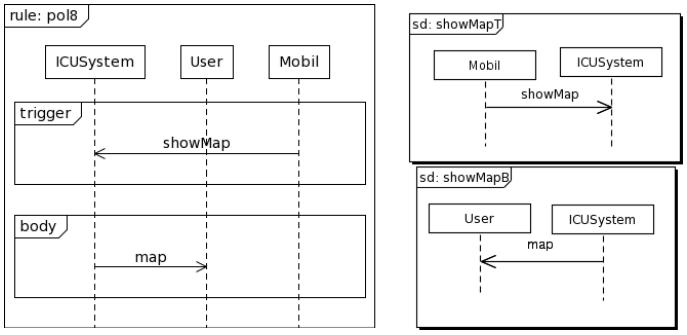


Figure C.8: One trigger message has another sender than in the system specification, but the body is present

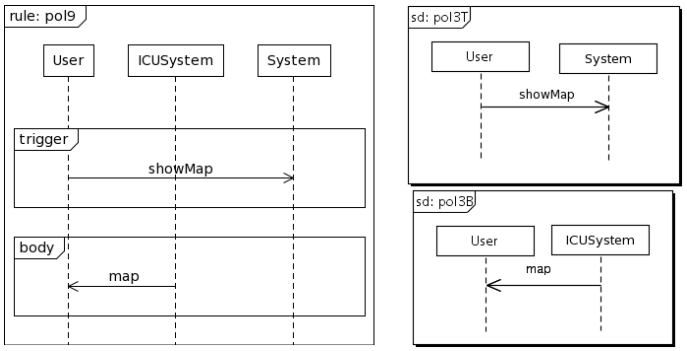


Figure C.9: One trigger message has another receiver than in the system specification, but the body is present

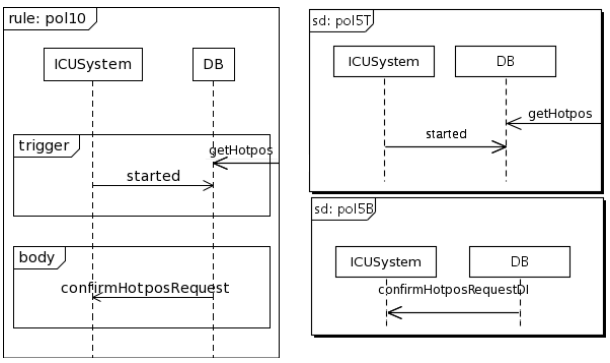


Figure C.10: The trigger messages are not in the same order as in the system specification, but the body is present

Appendix D

Evaluating the Køller-Method

We tested the policy specifications from Appendix C against the system specification, ICU, from Appendix B. We will here go through two of the tests, to give example of how we conducted the Køller-methods steps.

The Køller-method does not support any other operators then seq in the system specification. We therefore have defined the ICU-system with the use of only the seq-operator also in Appendix B.

D.1 Policy 1

We here go through the test with the policy specification named pol1, illustrated in Figure D.1 on page 112. As stated in Appendix C, we have used the word body instead of obligation, and prohibition. We have done it since we will test the policy with both types.

We first chose the lifeline to test, and in this case it was lifeline ICUSystem. Then we chose the part of the system specification that needed to be tested, and the part of the system specification is illustrated in Figure D.2 on page 113. There are three sequence diagrams that together tell the same as the sequence diagram illustrated in Figure B.2 on page 102.

We then started on Step 2, and began making the alphabets. First we made the alphabet of the system specifications, and then the alphabet of the policy specification. We show them both in Table D.1 on page 112.

When we made the policy specification's alphabet we started with the trigger, and then the body. For each message in the trigger, we checked if the message where present in the system specification's alphabet. As we can see from the Table D.1 on page 112, the trigger message begin is not present in the system specification's alphabet, and we then jumped to

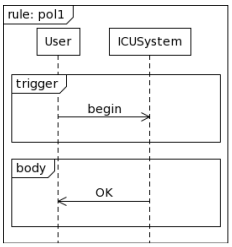


Figure D.1: The policy specification pol1

Diagram	Input	Output
System	start, register, registerOK, registerUsernameTaken, registerAlreadyRegisterd	registerID, registerOK1, registerUsernameTaken1, registerAlreadyRegisterd1
Policy	begin	OK

Table D.1: The alphabet of the system specification and policy specification used in this example

Step 4.

In Step 4 we wrote a report based on the test we had conducted. We knew that the trigger where not present, so therefore the system specification adhere to the policy specification. This both if the policy specification has the type obligation and prohibition.

D.2 Policy 2

We will also go through how we tested the policy specification, pol2, illustrated in Figure D.3 on page 114. The part of the system specification that we needed to test, is illustrated in Figure D.4 on page 114. Those sequence diagrams tell the same as the sequence diagram illustrated in Figure B.5 on page 103. We had one test with the system specification ICUhp, and one with ICUhp1, but the final result is based on both test results.

In this example we also chose lifeline ICUSystem, and started making the alphabet of the system specification and the policy specification. The alphabet of the system specification, ICUhp, and the policy specification, pol2, are showed in Table D.2.

From the table we saw that the trigger message disallowHotposDI is not present in the system specification's alphabet. We therefore did not need to test, and jumped to Step 4.

In Step 4 we made a report stating that the trigger message is not present, so the system specification, ICUhp, adhere to the policy specification, pol2.

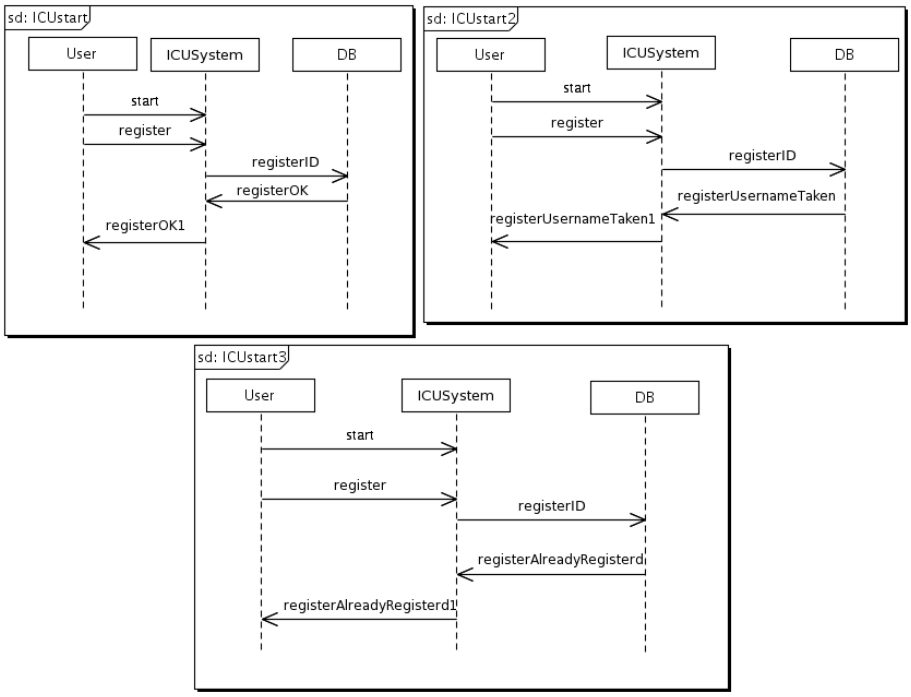


Figure D.2: The part of the ICU-system we want to test against pol1

Diagram	Input	Output
System	start, getHotposUI, hotpos	getHotposID, map
Policy	getHotposUI, disallowHotposDI	getHotposID, disallowHotposIU

Table D.2: The alphabet of the system specification, ICUhp, and policy specification pol2

For us to know if the whole system specification adhere to the policy specification we needed to test the other part, ICUhp1, as well. We chose the same lifeline, but needed to make a new alphabet. The alphabet of the system specification, ICUhp1, and the policy specification, pol2, are showed in Table D.3 on page 114.

This part of the system specification contained all the messages from the policy specification. We therefore went future in Step 2, and made the alphabets of the test diagrams. The test diagram's alphabets are showed in Table D.4 on page 115.

We then found the orders the messages in the alphabets of the test diagrams could have. The first test diagram could have this trace:
<?getHotposUI, !getHotposID, ?disallowHotposDI, !disallowHotposIU>
The second one could have these traces:

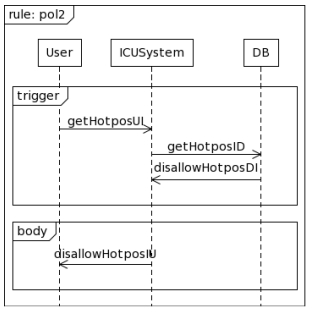


Figure D.3: The policy specification pol2

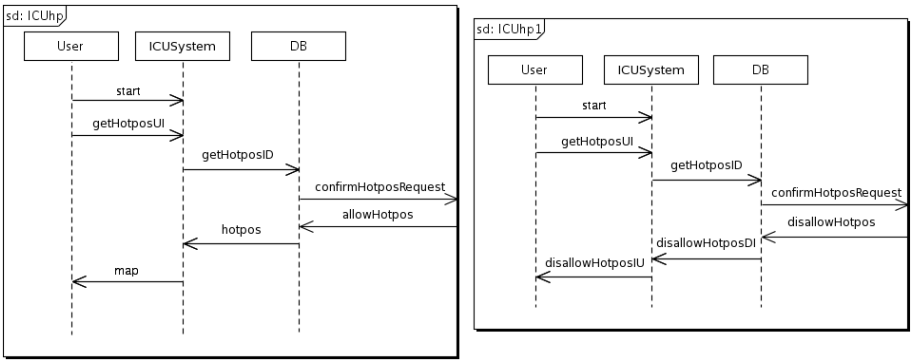


Figure D.4: The part of the ICU-system we want to test against pol2

<?getHotposUI, !getHotposID, ?disallowHotposDI, !disallowHotposIU, ?start>
<?getHotposUI, !getHotposID, ?disallowHotposDI, ?start, !disallowHotposIU>
<?getHotposUI, !getHotposID, ?start, ?disallowHotposDI, !disallowHotposIU>
<?getHotposUI, ?start, !getHotposID, ?disallowHotposDI, !disallowHotposIU>
<?start, ?getHotposUI, !getHotposID, ?disallowHotposDI, !disallowHotposIU>

The next step was to test the system specification against the test diagrams with the traces stated above. We first tested the system specification against the first test diagram,

Diagram	**Input**	**Output**
System	start, getHotposUI, disallowHotposDI	getHotposID, disallowHotposIU
Policy	getHotposUI, disallowHotposDI	getHotposID, disallowHotposIU

Table D.3: The alphabet of the system specification, ICUhp1, and policy specification pol2

TD	Input	Output
1	getHotposUI, disallowHotposDI	getHotposID, disallowHotposIU
2	start, getHotposUI, disallowHotposDI	getHotposID, disallowHotposIU

Table D.4: The alphabet of the test diagrams in this example

and got these results from the Escalator:

- pass: <!getHotposUI, ?getHotposUI>
- pass: <!start, ?start, !getHotposUI, ?getHotposUI, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>

When testing the system specification against the second test diagram we got these results:

- pass: <!start, ?start, !getHotposUI, ?getHotposUI, theta, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, theta, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>

We then had to start with Step 4, and analyze the results. The final result from the first test diagram, if the policy specification where an obligation, where:

- pass: <!getHotposUI, ?getHotposUI>

The final result from the second test diagram, if the policy specification where an obligation, where:

- pass: <!start, ?start, !getHotposUI, ?getHotposUI, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>
- pass: <!start, ?start, !getHotposUI, ?getHotposUI, theta, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, theta, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>

There where only pass-verdicts. That since the traces either did not contain the trigger, or contained both the trigger and body. So this results state that the system specification, ICUhp1, adheres to the policy specification, pol2.

If the policy specification where a prohibition we got this final result from the first test diagram:

- pass: <!getHotposUI, ?getHotposUI>

And from the second test diagram we got these final results:

- fail: <!start, ?start, !getHotposUI, ?getHotposUI, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>
- fail: <!start, ?start, !getHotposUI, ?getHotposUI, theta, !getHotposID, ?getHotposID, !disallowHotposDI, ?disallowHotposDI, theta, !disallowHotposIU, ?disallowHotposIU, theta>
- pass: <!getHotposUI, ?getHotposUI>

Here we got both pass, and fail-verdicts. That since there where traces without the trigger, and traces with both the trigger and body. The results stated that the system specification, ICUhp1, did not adhere to the policy specification, pol2. That since if there is one fail-verdict, it means that the system specification does not adhere to the policy specification, no matter how many pass-verdicts.

When we tested the policy specification pol2 with the type obligation, we got pass both when testing it together with the system ICUhp and ICUhp1. This means that the system specification, as far as this test says, adheres to the policy specification pol2 if it has the type obligation.

When we tested the policy specification pol2 with the type prohibition, we got different answers when testing it together with ICUhp and ICUhp1. When testing it together with ICUhp we got pass, and with ICUhp1 we got fail. As stated when there are different answers, the result is fail. This means that the system specification, as far as this test says, does not adhere to the policy specification pol2 if it has the type prohibition.

Appendix E

Evaluating the Køller-Tool

We tested the policy specifications from Appendix C against the system specification, ICU, from Appendix B. We will here go through two of the tests, to give examples of how the Køller-tool conduct its steps. The policy specifications will be the same as the ones we used when illustrating the evaluation of the Køller-method in Appendix D.

E.1 Policy 1

We here go through the test with the policy specification named pol1, illustrated in Figure E.1. As stated in Appendix C, we have used the word body instead of obligation, and prohibition. That is because we will test the policy with both types.

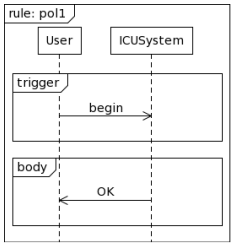


Figure E.1: The policy specification pol1

We chose the part of the system specification that needed to be tested, and the part of the system specification is illustrated in Figure E.2 on page 118. There are three sequence diagrams that together tell the same as the sequence diagram illustrated in Figure B.2 on page 102. We needed to conduct one test for each of the system diagrams. For all the system diagrams we wanted to test the lifeline ICUSystem.

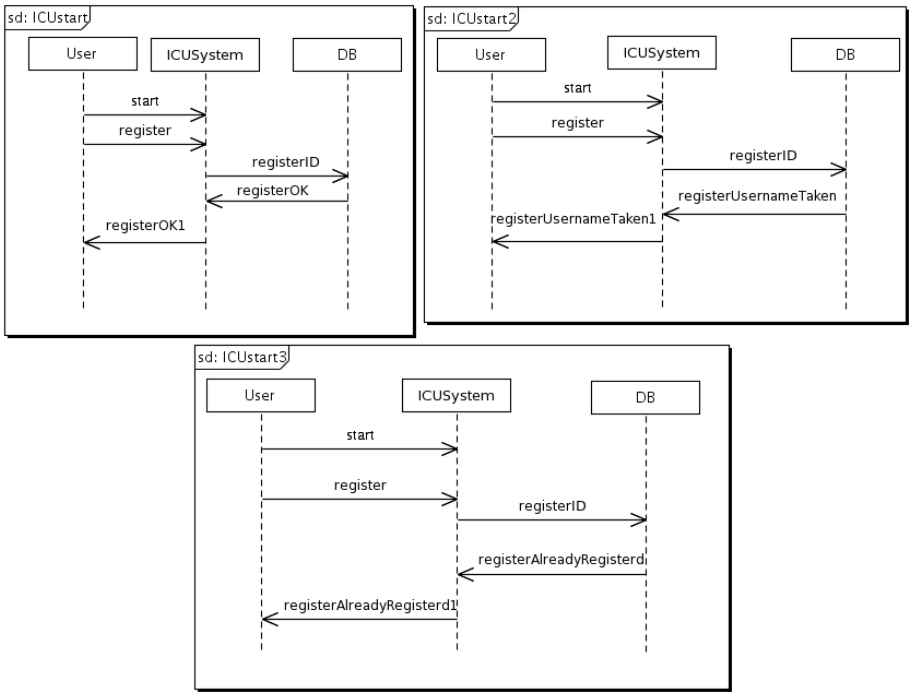


Figure E.2: The part of the ICU-system we want to test against pol1

In Figure E.3 we have illustrated how we filled the fields to the Køller-tool. We filled the fields the same way for all the three tests, but changed the name of the system diagram.

We tested all three sequence diagrams containing the system specification against the policy specification, with the type obligation, and got the following reports.

```
RESULT FROM TESTING /ICUSystem/system/ICUstart.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 281 ms, used not test diagrams
Result(s): PASS
The trigger message 'begin' is not present in the
system diagram. This means that the policy is being held.

RESULT FROM TESTING /ICUSystem/system/ICUstart2.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 0 ms, used not test diagrams
Result(s): PASS
```

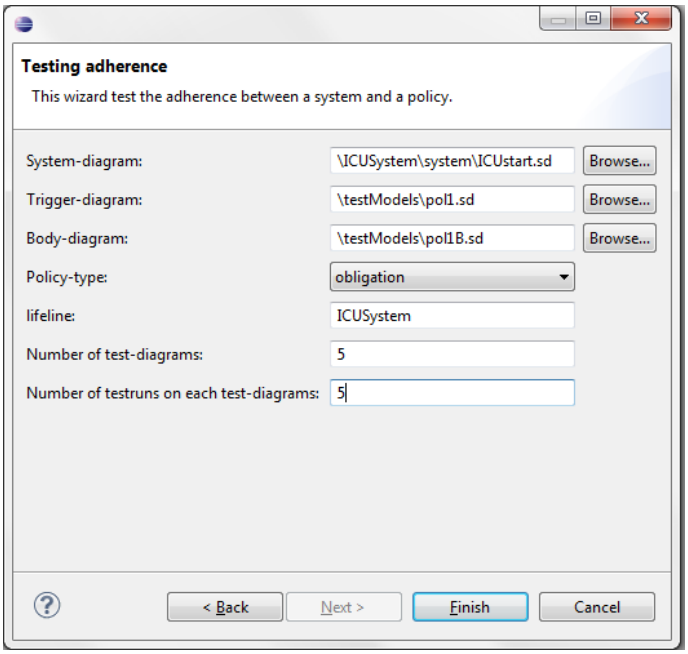


Figure E.3: The information we gave the Køller-tool when performing the policy adherence test

```
The trigger message 'begin' is not present in the
system diagram. This means that the policy is being held.

RESULT FROM TESTING /ICUSystem/system/ICUstart3.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 78 ms, used not test diagrams
Result(s): PASS
The trigger message 'begin' is not present in the
system diagram. This means that the policy is being held.
```

We did the same, but changed the policy rule to prohibition and got the following reports.

```
RESULT FROM TESTING /ICUSystem/system/ICUstart.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 10 ms, used not test diagrams
Result(s): PASS
The trigger message 'begin' is not present in the
```

```
system diagram. This means that the policy is being held.

RESULT FROM TESTING /ICUSystem/system/ICUstart2.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 10 ms, used not test diagrams
Result(s): PASS
The trigger message 'begin' is not present in the
system diagram. This means that the policy is being held.

RESULT FROM TESTING /ICUSystem/system/ICUstart3.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/testModels/pol1.sd AND BODYFILE /testModels/pol1B.sd
The testing took 10 ms, used not test diagrams
Result(s): PASS
The trigger message 'begin' is not present in the
system diagram. This means that the policy is being held.
```

All the tests, with both types of policies, gave the result pass. We got pass since the trigger where not present in the system specification, so the policy specification, pol1, does not apply to the system specification, ICU, as far as our tests tells us.

E.2 Policy 2

We here go through the test with the policy specification named pol2, illustrated in Figure E.4.

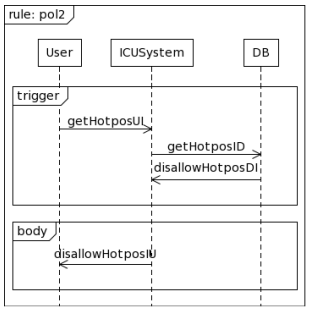


Figure E.4: The policy specification pol2

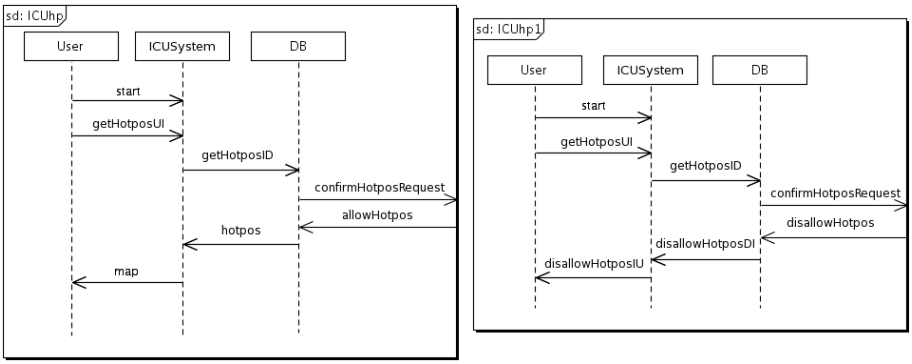


Figure E.5: The part of the ICU-system we want to test against pol2

We chose the part of the system specification that needed to be tested, and the part of the system specification is illustrated in Figure E.5. There are two sequence diagrams that together tell the same as the sequence diagram illustrated in Figure B.5 on page 103. We needed to conduct one tests for each of the system diagrams. For both the system diagrams we wanted to test the lifeline ICUSystem.

We filled the fields to the Køller-tool in the same manner as in the example of pol1, and chose the policy rule obligation. After testing both sequence diagrams containing the system specification we got the following reports.

```
 RESULT FROM TESTING /ICUSystem/system/ICUhp.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/ICUSystem/policy/dhprT.sd AND BODYFILE
/ICUSystem/policy/dhprB.sd
The testing took 60 ms, used not test diagrams
Result(s): PASS
The trigger message 'disallowHotposDI' is not present
in the system diagram. This means that the policy is
being held.

 RESULT FROM TESTING /ICUSystem/system/ICUhp1.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/ICUSystem/policy/dhprT.sd AND BODYFILE
/ICUSystem/policy/dhprB.sd
The testing took 6074 ms, used test diagrams
Result(s): PASS
Pass - trigger is not present
```

```
   <, theta>
Pass - trigger is not present
   <>
Pass - both trigger and body are present
   <, theta, !start, ?start, !getHotposUI,
?getHotposUI, !getHotposID, ?getHotposID, theta,
!disallowHotposDI, ?disallowHotposDI,
!disallowHotposIU, ?disallowHotposIU, theta>
Pass - both trigger and body are present
   <, theta, !start, ?start, !getHotposUI,
?getHotposUI, !getHotposID, ?getHotposID,
!disallowHotposDI, ?disallowHotposDI,
!disallowHotposIU, ?disallowHotposIU, theta>
Pass - both trigger and body are present
   <!start, ?start, theta, !getHotposUI,
?getHotposUI, !getHotposID, ?getHotposID,
!disallowHotposDI, ?disallowHotposDI,
!disallowHotposIU, ?disallowHotposIU, theta>
```

Both tests gave the result pass. We got pass since the traces either did not contain the trigger, or contained both the trigger and body. From this tests we assume that the policy specification, pol2, adheres to the system specification, ICU.

When testing with the policy rule prohibition we got the following reports.

```
RESULT FROM TESTING /ICUSystem/system/ICUhp.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/ICUSystem/policy/dhprT.sd AND BODYFILE
/ICUSystem/policy/dhprB.sd
The testing took 10 ms, used not test diagrams
Result(s): PASS
The trigger message 'disallowHotposDI' is not present
in the system diagram. This means that the policy is
being held.
```

```
RESULT FROM TESTING /ICUSystem/system/ICUhp1.sd
AGAINST THE POLICY DEFINED IN THE TRIGGERFILE
/ICUSystem/policy/dhprT.sd AND BODYFILE
/ICUSystem/policy/dhprB.sd
```

```
The testing took 5630 ms, used test diagrams
Result(s): PASS, FAIL
PASS - did not find both body and trigger
   <, theta>
PASS - did not find both body and trigger
   <>
FAIL - both trigger and body are present
   <, theta, !start, ?start, theta, !getHotposUI,
?getHotposUI, !getHotposID, ?getHotposID,
!disallowHotposDI, ?disallowHotposDI, !disallowHotposIU,
?disallowHotposIU, theta>
FAIL - both trigger and body are present
   <!start, ?start, theta, !getHotposUI, ?getHotposUI,
!getHotposID, ?getHotposID, theta, !disallowHotposDI,
?disallowHotposDI, !disallowHotposIU, ?disallowHotposIU,
theta>
FAIL - both trigger and body are present
   <, theta, !start, ?start, theta, !getHotposUI,
?getHotposUI, !getHotposID, ?getHotposID, theta,
!disallowHotposDI, ?disallowHotposDI, !disallowHotposIU,
?disallowHotposIU, theta>
FAIL - both trigger and body are present
   <!start, ?start, !getHotposUI, ?getHotposUI, !getHotposID,
?getHotposID, theta, !disallowHotposDI, ?disallowHotposDI,
!disallowHotposIU, ?disallowHotposIU, theta>
```

The two tests did give different results. The first one concluded on pass, and the second one on both pass and fail. As we have stated when there is one fail-verdict, no matter how many pass-verdicts, the final result is fail. So based on this tests we assume that the policy specification, pol2, does not adhere to the system specification, ICU.